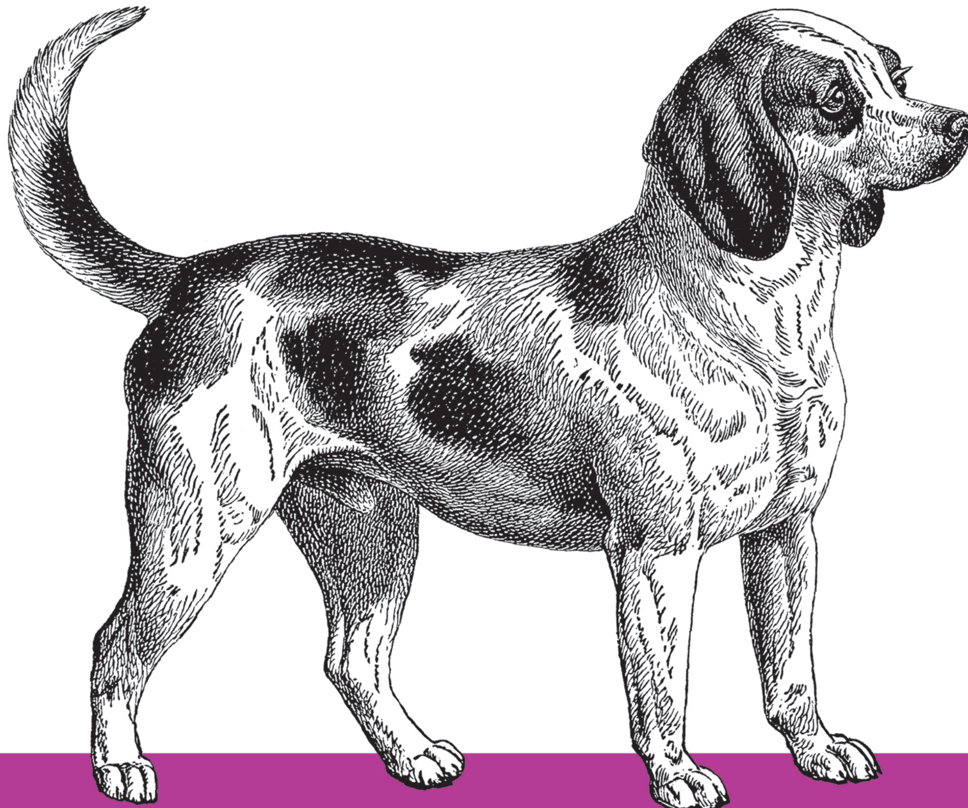
O'REILLY®

TURING 图灵程序设计丛书



GitHub实践

Building Tools with GitHub

活学活用GitHub，轻松构建属于你的软件工具

[美] Chris Dawson Ben Straub 著

安道 译

中国工信出版集团 人民邮电出版社 POSTS & TELECOM PRESS

TURING 图灵程序设计丛书

GitHub实践

Building Tools with GitHub

Customize Your Workflow

[美] Chris Dawson Ben Straub 著
安道 译

O'REILLY®

Beijing • Cambridge • Farnham • Köln • Sebastopol • Tokyo

O'Reilly Media, Inc.授权人民邮电出版社出版

人民邮电出版社
北 京

图书在版编目（C I P）数据

GitHub实践 / (美) 克里斯•道森 (Chris Dawson) ,
(美) 本•斯特劳勃 (Ben Straub) 著 ; 安道译. -- 北
京 : 人民邮电出版社, 2017.1
（图灵程序设计丛书）
ISBN 978-7-115-43806-5

Ⅰ. ①G… Ⅱ. ①克… ②本… ③安… Ⅲ. ①软件工
具－程序设计 Ⅳ. ①TP311.56

中国版本图书馆CIP数据核字(2016)第249169号

内 容 提 要

本书一窥 GitHub 内部使用的工具，以叙事的方式描述构建软件元工具的相关过程，其中不只介绍相关的技术，还会说明折中方案、重构的现实意义，以及编写元工具所面临的挑战。对已经熟悉 Git 或 GitHub、想提升相关技能的读者，书中介绍了如何使用 GitHub API 及相关的开源技术，如 Jekyll（网站生成工具）、Hubot（NodeJS 聊天机器人）和 Gollum（维基）构建工具。

本书适合所有想要使用 GitHub 进行开发的程序员。

◆ 著　　[美] Chris Dawson　Ben Straub
译　　安　道
责任编辑　朱　巍
执行编辑　温　雪　魏书莉
责任印制　彭志环

◆ 人民邮电出版社出版发行　　北京市丰台区成寿寺路11号
邮编　100164　　电子邮件　315@ptpress.com.cn
网址　http://www.ptpress.com.cn
北京　　　　印刷

◆ 开本：700×1000　1/16
印张：15.75
字数：372千字　　　　2017年 1 月第 1 版
印数：1 – 4 000册　　　　2017年 1 月北京第 1 次印刷
著作权合同登记号　图字：01-2016-6550号

定价：69.00元

读者服务热线：(010)51095186转600　印装质量热线：(010)81055316

反盗版热线：(010)81055315

广告经营许可证：京东工商广字第 8052 号

O’Reilly Media, Inc.介绍

O’Reilly Media 通过图书、杂志、在线服务、调查研究和会议等方式传播创新知识。自 1978 年开始，O’Reilly 一直都是前沿发展的见证者和推动者。超级极客们正在开创着未来，而我们关注真正重要的技术趋势——通过放大那些“细微的信号”来刺激社会对新科技的应用。作为技术社区中活跃的参与者，O’Reilly 的发展充满了对创新的倡导、创造和发扬光大。

O’Reilly 为软件开发人员带来革命性的“动物书”；创建第一个商业网站（GNN）；组织了影响深远的开放源代码峰会，以至于开源软件运动以此命名；创立了 Make 杂志，从而成为 DIY 革命的主要先锋；公司一如既往地通过多种形式缔结信息与人的纽带。O’Reilly 的会议和峰会集聚了众多超级极客和高瞻远瞩的商业领袖，共同描绘出开创新产业的革命性思想。作为技术人士获取信息的选择，O’Reilly 现在还将先锋专家的知识传递给普通的计算机用户。无论是通过书籍出版、在线服务或者面授课程，每一项 O’Reilly 的产品都反映了公司不可动摇的理念——信息是激发创新的力量。

业界评论

“O’Reilly Radar 博客有口皆碑。”

——*Wired*

“O’Reilly 凭借一系列（真希望当初我也想到了）非凡想法建立了数百万美元的业务。”

——*Business 2.0*

“O’Reilly Conference 是聚集关键思想领袖的绝对典范。”

——*CRN*

“一本 O’Reilly 的书就代表一个有用、有前途、需要学习的主题。”

——*Irish Times*

“Tim 是位特立独行的商人，他不光放眼于最长远、最广阔的视野，并且切实地按照 Yogi Berra 的建议去做了：‘如果你在路上遇到岔路口，走小路（岔路）。’回顾过去，Tim 似乎每一次都选择了小路，而且有几次都是一闪即逝的机会，尽管大路也不错。”

——*Linux Journal*

目录

前言

本书说明如何构建软件工具。

如果你经常编写软件，肯定知道这其实是创建工具的过程。软件也是工具。说到底，电子表格只是加减数字的工具，电子游戏只是打发无聊时间的工具。在人们开始编写软件工具之后，我们几乎立即发现，为了顺利写出一开始想写的那个软件，需要有更多的工具来支持。我们把这些支持软件开发的工具（不是常规意义上的软件工具）叫作元工具（meta-tool）。

在软件开发领域，最重要的一个元工具是 Git。这个元工具能帮助软件开发者解决编写软件过程中的复杂问题。使用 Git，软件开发者可以存储程序的快照（如果需要，可以轻易还原快照），还便于与其他程序员协作（这是个相当复杂的问题）。Git 是一种源码管理（Source Code Management，SCM）工具，虽然在此之前有很多 SCM 工具，但是都没像 Git 这样对软件领域产生如此大的影响。目前，Git 在 SCM 工具中处于领先地位。

GitHub 公司及早发现了 Git 的无穷潜力，以 Git 的功能为底层基础，构建了一层 Web 服务。不难想象，GitHub 成功的因素之一是，员工们从一开始就积极跟随潮流，编写大量的元工具。开发元工具要耐得住寂寞，不能急于求成。GitHub 的员工们对这种做事顺序引以为荣，并撰写了大量文章说明了这样做的好处，例如便于新员工融入，工作流程对所有员工都透明。

本书一窥 GitHub 内部使用的工具。GitHub.com 网站本身也是一个元工具，我们会从多个方面讨论 GitHub 服务。具体而言，我们涉及的技术有 GitHub API 及相关的 GitHub 技术、Gollum 维基、静态页面生成工具 Jekyll，以及聊天机器人 Hubot（如果你对这些都不熟悉，不用担心，我们将在不同的章节详细说明）。

我要重申一点，这不是上述几项技术的参考书。这是一本故事书，以叙事的方式描述构建软件元工具的相关过程，期间不只介绍相关的技术，还会说明折中方案、重构的现实意

义，以及编写元工具面临的挑战。

编写元工具所需的思维与常规的软件不同。一般来说，元工具是开源的，用法和责任都不同寻常。有人可能认为，与普通用户相比，软件工程师对质量有更高的要求，因为软件不能满足软件开发者的需求时，他们可以改进或派生。编写元工具需要付出更多的精力，因而自动化测试几乎是必备的。开发元工具之前一定要知道这些概念，本书将告诉你如何在构建元工具的过程中运用这些概念。

为什么使用API，为什么选择GitHub API

使用 API 支持应用是当下常见的做法，这也是应用开发的发展方向。如今显示设备多种多样，使用 API 能在不同的设备中更好地呈现数据。以远程服务 API 支持的应用，起初可能是运行在 Apple iOS 操作系统中的移动应用。经市场验证之后，如果发现这种商业模式不正确，可以迅速响应变化的需求，为 Android 穿戴设备构建新应用，构建车载应用，或者其他控制台（或非控制台）应用。只要应用能调用远程 API 收发数据，你就能为任何平台构建任何用户界面。

此外，我们自己也可以编写和托管 API。Ruby、Go 和 Java 等流行语言的很多框架都支持使用标准的架构风格（如 REST）构建 API。不想自己构建的话，还可以使用第三方 API。本书专门针对一个第三方 API，即 GitHub API。

为什么是 GitHub API 呢？对构建软件的你来说，基本上离不开 GitHub API，因为你可能就使用 GitHub 管理软件代码。即便不用 GitHub，也有可能使用 Git。而 GitHub API 也能操作 Git，因为它把 Git 的功能抽象成网络编程接口了。

GitHub API 算是我见过设计得最好的 API 了。它是超媒体 API，基本上成功解决了一个棘手问题：让 API 客户端适应 API 的变化。GitHub API 有良好的版本划分，功能完整，而且实现了大多数 Git 功能。GitHub API 由多个结构合理的部分组成，便于构建应用。GitHub API 是个典型，通过它可以学习好的 API 应该怎么设计。

本书结构

GitHub API 的功能非常全面，通过它可以访问和修改 Git 仓库中存储的或与之有关的几乎所有数据和元工具。根据 GitHub API 的文档（https://developer.github.com/v3/），我按照字母顺序简要列出了其各部分的作用。

- Activity：开发者关注的事件通知
- Gists：以编程的方式创建和分享代码片段
- Git Data：通过远程 API 访问原始的 Git 数据

- Issues：添加和修改工单
- Miscellaneous：无法纳入其他分类的 API
- Organizations：存取组织成员数据
- Pull Requests：一个强大的 API，建构在合并过程之上
- Repositories：修改与仓库有关的一切数据
- Search：在整个 GitHub 数据库中搜索代码
- Users：访问用户数据
- Enterprise：企业内部的 GitHub 专用的 API

此外，有些重要的技术在 GitHub API 文档中没有说明。虽然这些技术不是 API 的一部分，但是使用 GitHub 时应该知道。

- Jekyll 和“gh-pages”：托管博客和静态文档
- Gollum：与仓库关联的维基
- Hubot：GitHub 广泛使用的可编程聊天机器人

GitHub 技术栈的各个部分分别在不同的章节介绍（有两个不会介绍，原因如后）。GitHub API 文档是优秀的参考，编写与 GitHub API 通信的应用时会经常查阅。本书的目的则不同，我们将叙述如何使用 GitHub 提供的技术构建应用。这些故事会告诉你使用 GitHub API 时要采取哪些折中措施，有什么注意事项。如果需要，一章可能涵盖多个 API。通常，每一章都尽量专注于一个主要的 API，努力不涉及其他 API，但是大多数章节确实需要涉及别的 API。

下面概述各章的内容。

第 1 章首次介绍如何使用命令行 HTTP 客户端 cURL 访问 API，并说明响应的格式、如何在命令行中解析响应，以及身份验证方式。这是本书唯一没有使用上述技术构建应用的一章。

第 2 章介绍 Gist API、几个命令行工具，以及 Ruby 语言 API 客户端 Octokit。然后，使用 Gist API 构建一个简单的 Ruby 服务器，存储并显示 Gist。

第 3 章说明如何使用 Gollum 命令行工具及相关的 Ruby 库（gem）。Gollum 由 Grit 提供支持，Grit 是一个 C 语言绑定，用于访问 Git 仓库。我们还会说明 Git 存储格式的一些细节，在 Git 仓库中存储大文件的方式，以及如何使用 Git 命令行工具访问这些信息。这一章使用 Gollum 和 Grit 库构建一个图像管理工具，不过它仍是一个常规的 Gollum 维基，可以发布到 GitHub 中。

第 4 章探讨 Search API，然后使用 Python 构建一个 GUI 工具，用于搜索 GitHub 中的仓库。

第 5 章介绍 GitHub API 中一个相对新的部分，这一部分用于描述第三方工具与代码之间的交互。这一章使用 C# 和 .NET GitHub API 库 Nancy 构建一个应用。

如果把按照特定方式组织的仓库推送到 GitHub 中，GitHub 会将其转换成功能完整的博客，基本上相当于 WordPress 网站（当然，没那么复杂）。第 6 章说明仓库的文件结构，如何在 Jekyll 网站中使用 Markdown，如何使用 Liquid 模板引擎提供的循环结构，以及如何使用 Ruby 语言开发一个工具，从 Internet Archive 中把整个网站导出到 Jekyll 网站中。这一章展示如何使用缓存爬取网站。使用 API 或第三方公开信息时缓存很有用。

第 7 章为 Android 操作系统开发一个移动应用。这个应用使用 GitHub API 的 Git Data 部分读取信息，再把信息写入一个 Jekyll 网站仓库。这一章展示如何使用 UI 测试工具 Calabash 为 Android 应用编写用户界面测试，验证 GitHub API 的响应。

Hubot 是一个 JavaScript（NodeJS）聊天机器人，技术人员使用它可以更进一步，由“DevOps”（开发运维）变成“ChatOps”（聊天运维）。第 8 章说明如何使用 GitHub API 的 Activity 和 Pull Requests 部分。此外，还说明如何模拟 GitHub 的通知，以及如何编写可测试的 Hubot 扩展（JavaScript 代码通常难以测试）。我们将把这些串在一起，构建一个机器人，让它自动分配拉取请求审查邀请。

你知道可以把整个“单页应用”托管在 GitHub 中吗？第 9 章说明如何使用 JavaScript 语言构建一个咖啡店信息应用，这个应用的数据库是普通的文件，托管在 GitHub 中。更重要的是，我们将展示如何编写可测试的 JavaScript 应用，说明如何模拟所需的 GitHub API。

本书不会介绍 Organizations API，因为这是 GitHub API 的一小部分，作用简单，只能列出组织，以及修改组织的元数据。学会 GitHub API 的其他部分后，这个不重要的部分自然就会用了。

本书也不会介绍 Users API。你可能觉得这一部分很重要，其实不然，Users API 只提供了一个端点，用于列出用户的信息、添加或删除 SSH 密钥、修改电子邮件地址和修改关注者。

本书没有为工单专开一章。以前，GitHub 把工单和拉取请求放在 API 的同一部分中，不过拉取请求越来越重要，所以 GitHub API 文档把它单独列为一部分。其实，在 GitHub 内部，二者仍然存储在同一个数据库中，而且拉取请求目前还是工单的一种。第 8 章将说明如何使用拉取请求，工单可以比照使用。

Enterprise API 几乎与 GitHub.com 网站的 API 一样，本书没有单写一章介绍企业版 API 的用法，不过在附录 B 中举了几个示例，说明如何使用企业版。本书还为各章使用的不同语言提供了企业版句法，这样书中的各个示例就能在企业版中使用了。

本书通过这些故事讲述如何使用 GitHub 背后的技术，希望借此让你一窥使用 GitHub API

构建应用的开发者是如何思考问题的。

本书的目标读者

本书应该能为已经使用过 Git 或 GitHub 并想提升技能的读者提供有用的信息。没有用过 GitHub 或 Git 的读者应该先读一本入门书。[1]

你应该至少熟悉一门现代的命令式编程语言。本书不要求你是专家级程序员，但是要有一定的编程经验，至少熟悉一门编程语言。

你应该知道 HTTP 协议的基础知识。GitHub 团队设计 API 时采用了十分标准的 REST 式架构。你应该知道 GET 请求和 POST 请求的区别，至少要知道 HTTP 状态码的意思。

如果熟悉其他 Web API，阅读本书会轻松一些。本书的目标是告诉你如何使用架构、设计和测试良好的 API，构建有趣且强大的工具。如果你没怎么用过 Web API，但是用过其他类型的 API，这样也行。

你将学到什么

本书主要介绍 GitHub 和强大的 GitHub API 提供的功能。不要以为这会限制你对 Git 的使用。假如你是 Android 开发者，使用 Git 管理应用的源码，如果你想在其他地方使用 Git，没问题，本书能打开你的视野，教你 Git 和 GitHub 的高级用法。如果你在自己的项目中已经使用 Git，想在更大的群体中推广 Git，本书会教你 GitHub 引领的"社交编程"风潮。本书为使用其他分布式版本控制系统的软件开发者架起了桥梁，能指引他们转用 Git 和 GitHub 这样的 Web 服务。

资深开发者都喜欢自动化工具。本书提供了一些示例，说明如何把单调的任务转换成可自动和重复执行的过程。这些示例使用多门编程语言编写，让你了解如何与 GitHub API 通信。

为了让本书适合更多的人群阅读，我们没有限定必须使用哪个编辑器或操作系统，很多示例程序都在命令行中执行。如果你不熟悉命令行，本书能让你深入了解它的用法，相信你读完本书之后会发现命令行的强大。如果从 5 岁起你的爸爸就强迫你使用命令行，导致你对它充满恨意，那么通过本书你将重新爱上 bash shell。

如果你能透过 GitHub 提供的技术看到其背后文化和思想的改变，很有可能会发现一种符合现代社会的工作方式。本书集中精力讨论工具本身，工具能做什么则由你自己去探索。

书中各章几乎都有对应的仓库，这些仓库托管在 GitHub 中，供你查看书中讨论的代码。

注 1： 可参考《GitHub 入门与实践》，人民邮电出版社出版。——编者注

你可以随意派生这些示例，在自己的项目和工具中使用。

最后，本书会教你编写可测试的 API 后端代码。即便是最有经验的开发者，往往也觉得为代码编写测试是件有挑战的事，何况还要把测试代码写得像文学作品那样高深莫测。测试以 API 为后端的程序尤其艰难。我们要跳出单元测试的思维，以不同的方式思考。为了帮你跨过这道障碍，本书会告诉你怎么让与 GitHub API 交互的代码变得易于测试。

GitHub钟爱的语言

有两门语言与 GitHub 密不可分，你要安装并使用它们才能更好地理解本书的内容。

- Ruby
  这是一门简单、易读的编程语言，GitHub 公司的创始人在公司早期广泛使用。

- JavaScript
  这是唯一普遍使用的浏览器端编程语言，随着 NodeJS 的出现，其重要性迈上了新的台阶，甚至与开发 Web 应用的服务器端框架 Ruby on Rails 平起平坐了。独立开发者尤其喜爱该语言。

不可否认，本书的很多读者都已经熟悉 Ruby 或 JavaScript/NodeJS 了，因此正文没有说明它们的基本用法和安装方式，而是放到了附录中。不过，附录也没有介绍这两门语言的句法，我们希望你有使用其他语言的经验，能读懂任何命令式语言编写的代码。书中各章讨论 GitHub API 的方方面面，偶尔会讲解语言细节，但是不管你熟悉哪门语言，书中的代码应该都通俗易懂。附录讨论了这两门语言在 GitHub 发展过程中的作用，还着重说明了一些特殊文件的作用和安装方式。

安装和使用这两门语言不会浪费你的时间，相反，能为你建立坚实的基础，以便深入探索 GitHub API。书中有几章使用 Ruby 或 JavaScript，所以花点时间学习基本句法有助于更好地理解书中的内容。

对操作系统的要求

本书是两位作者在 MacBook Pro 中撰写的。MacBook 中都有 shell（“BASH”），与任何 Linux 设备中的 shell 几乎一样。如果你使用这两个操作系统，各章的代码都能顺利运行。

如果你使用 Windows 设备（或者没有 BASH shell 的操作系统），部分命令和代码示例可能无法运行，需要安装额外的软件才行。

简单的解决方法是使用 VirtualBox 和 Vagrant。VirtualBox 是免费的虚拟系统，能在 x86 硬件中使用。Vagrant 是开发环境管理工具。使用 VirtualBox 和 Vagrant 能快速安装 Linux

虚拟机。如果想安装这两个工具，分别访问 VirtualBox（https://www.virtualbox.org/wiki/Downloads）和 Vagrant（https://www.vagrantup.com/downloads.html）的下载页面。安装好之后，可以使用下述两个命令安装 Ubuntu 虚拟机：

```
$ vagrant init hashicorp/precise32
$ vagrant up
```

不适合阅读本书的读者

本书探讨 GitHub API 的用法，没有限定于单一的语言，而是使用多门不同的语言。本书不仅说明了 GitHub 团队设计 API 的方式，还分析了不同编程语言和社区对客户端库的实现方式。我们相信这样能教你更多知识，因此，如果你只对某一门语言感兴趣，想知道如何使用那门语言与 GitHub API 交互，那么这本书就不适合你。

本书努力证明 API 驱动的代码可以测试，而且这么做是有好处的。但是本书不是测试编写手册，不会教你怎么写出最好的测试代码。我们使用多门语言，为的是不加入各个社区对测试框架的争论。我们认识到多数软件项目完全没有测试，因此本书的重点是帮你跨过这只大拦路虎。如果你从未写过测试，那么要转换一下思维方式。我们希望通过具体的示例让你（尤其是没有写过测试的读者）知道如何为使用 API 的代码编写测试。有些章节对应的仓库中有更详细的测试组件，书中的测试大都简略，没有涵盖全部边缘情况。

排版约定

本书使用了下述排版约定。

- *楷体*
  表示新术语。

- 等宽字体（`constant width`）
  表示程序片段，以及正文中出现的变量、函数名、数据库、数据类型、环境变量、语句和关键字等。

- 加粗等宽字体（**`constant width bold`**）
  表示应该由用户输入的命令或其他文本。

- 斜体等宽字体（*`Constant width italic`*）
  表示应该替换成用户提供的值，或者由上下文决定的值。

这个图标表示一般性说明。

这个图标表示警告或提醒。

使用代码示例

补充材料（代码示例、练习等）可以从 https://github.com/xrd/building-tools-with-github 下载。

本书是要帮你完成工作的。一般来说，如果本书提供了示例代码，你可以把它用在你的程序或文档中。除非你使用了很大一部分代码，否则无需联系我们获得许可。比如，用本书的几个代码片段写一个程序就无需获得许可，销售或分发 O'Reilly 图书的示例光盘则需要获得许可；引用本书中的示例代码回答问题无需获得许可，将书中大量的代码放到你的产品文档中则需要获得许可。

我们很希望但并不强制要求你在引用本书内容时加上引用说明。引用说明一般包括书名、作者、出版社和 ISBN。比如："*Building Tools with GitHub* by Chris Dawson and Ben Straub (O'Reilly). Copyright 2016 Chris Dawson and Ben Straub, 978-1-491-93350-3."

如果你觉得自己对示例代码的用法超出了上述许可的范围，欢迎你通过 permissions@oreilly.com 与我们联系。

Safari® Books Online

Safari Books Online（http://www.safaribooksonline.com）是应运而生的数字图书馆。它同时以图书和视频的形式出版世界顶级技术和商务作家的专业作品。技术专家、软件开发人员、Web 设计师、商务人士和创意专家等，在开展调研、解决问题、学习和认证培训时，都将 Safari Books Online 视作获取资料的首选渠道。

对于组织团体、政府机构和个人，Safari Books Online 提供各种产品组合和灵活的定价策略。用户可通过一个功能完备的数据库检索系统访问 O'Reilly Media、Prentice Hall Professional、Addison-Wesley Professional、Microsoft Press、Sams、Que、Peachpit

Press、Focal Press、Cisco Press、John Wiley & Sons、Syngress、Morgan Kaufmann、IBM Redbooks、Packt、Adobe Press、FT Press、Apress、Manning、New Riders、McGraw-Hill、Jones & Bartlett、Course Technology 以及其他几十家出版社的上千种图书、培训视频和正式出版之前的书稿。要了解 Safari Books Online 的更多信息，我们网上见。

联系我们

请把对本书的评价和问题发给出版社。

美国：

> O'Reilly Media, Inc.
> 1005 Gravenstein Highway North
> Sebastopol, CA 95472

中国：

> 北京市西城区西直门南大街 2 号成铭大厦 C 座 807 室（100035）
> 奥莱利技术咨询（北京）有限公司

O'Reilly 的每一本书都有专属网页，你可以在那儿找到本书的相关信息，包括勘误表、示例代码以及其他信息。本书的网站地址是：

> http://shop.oreilly.com/product/0636920043027.do

对于本书的评论和技术性问题，请发送电子邮件到：bookquestions@oreilly.com

要了解更多 O'Reilly 图书、培训课程、会议和新闻的信息，请访问以下网站：

> http://www.oreilly.com

我们在 Facebook 的地址如下：http://facebook.com/oreilly

请关注我们的 Twitter 动态：http://twitter.com/oreillymedia

我们的 YouTube 视频地址如下：http://www.youtube.com/oreillymedia

致谢

Chris 要感谢他可爱的妻子 Nicole。与我相处和持家的过程中你充满了智慧，真希望我能把这些智慧用到本书中。我的儿子 Roosevelt 活力四射，不断感染着我，即使我已筋疲力尽，也能给我前进的动力。感谢我的女儿 Charlotte，你像微笑的佛像那样让我安宁。感谢我的母亲，谢谢你教我写作，最重要的是，告诉我为什么写作，这是技术圈急需的技能。感谢 Tim O'Brien 邀请我参与这个项目，谢谢你，希望以后还有机会合作。感谢 Bradley

Horowitz，你证明了小小的善举可以产生极大的影响。还要感谢 David J. Groom，虽然我们从未谋面，但是你的建议和鼓舞对这本书极其重要，你的鼓励让我写完了这本书，如此才能与某一天阅读本书的读者结识。

Ben 要感谢他的妻子 Becky，谢谢她一直以来的支持，并在需要的时候督促他。没有她，世界将变得不完美。

电子书

扫描如下二维码，即可购买本书电子版。

第 1 章

开放的GitHub API

本章开始介绍如何使用 GitHub API 读写数据。后续的章节说明如何使用不同的客户端库通过 GitHub API 访问信息。这些客户端库故意隐藏 API 的具体细节，为你提供简洁且符合习惯的方法，用于访问托管在 GitHub 中的 Git 仓库，查看和修改里面的数据。不过，本章直接分析 GitHub API，详细说明原始 HTTP 请求和响应。本章还会讨论访问 GitHub 中公开和隐私数据的不同方式，并指出各自的不足之处。此外，本章会概述网络受限时如何在 Web 浏览器中访问 GitHub 的数据。

1.1 cURL

有时，你可能想立即通过 GitHub API 访问信息，而不想编写正式的程序；有时，你可能想立即获取 HTTP 原始请求的首部和内容；有时，你甚至可能会对客户端库的实现有疑惑，需要换个角度确认客户端库的行为是否正确。遇到这些情况时，最适合使用 cURL 这个简单的命令行 HTTP 工具。与最优秀的 Unix 工具一样，cURL 是个小型程序，功能十分专一，而且故意做了限制，只用于访问 HTTP 服务器。

cURL 与它熟谙的 HTTP 协议一样是无状态的。后面有一章会探讨这一局限性的解决方法，不过要注意，cURL 最适合用于发起一次性请求。

安装 cURL

大多数 OS X 设备中通常都安装了 cURL，在 Linux 中可以使用包管理器轻易安装（执行 `apt-get install curl` 或 `yum install curl` 命令）。如果使用的是 Windows，或者想自己动手安装，请访问 http://curl.haxx.se/download.html。

下面来发起一个请求。我们从 GitHub API 最基本的端点入手，地址是 https://api.github.com。

```
$ curl https://api.github.com
{
  "current_user_url": "https://api.github.com/user",
  "current_user_authorizations_html_url":
  "https://github.com/settings/connections/applications{/client_id}",
  "authorizations_url": "https://api.github.com/authorizations",
  "code_search_url":
  "https://api.github.com/search/code?q={query}{&page,per_page,sort,order}",
  "emails_url": "https://api.github.com/user/emails",
  "emojis_url": "https://api.github.com/emojis",
  ...
}
```

为了便于阅读，此处省略了部分响应。有几点需要特别注意：响应中有大量指向附属信息的 URL，URL 中包含参数，此外响应的格式是 JSON。

我们从这个 API 的响应中能得知什么呢？

1.2　列举API路径

GitHub API 是超媒体 API。超媒体的构成需要用一整本书才能说清楚（推荐阅读《使用 HTML5 和 Node 构建超媒体 API》），不过通过分析响应，我们能掌握超媒体的多数核心概念。首先，从前面那个 API 的响应可以看出，响应中包含一个映射，列出了接下来可能会发起请求的地址。当然，不是所有客户端都会使用这个信息，但是超媒体 API 的目标之一，是让客户端在不重新编写代码的前提下动态调整所用的端点。如果你觉得编写客户端时要考虑自动处理新端点，以防 GitHub 更改 API 这一点难以理解，不用太过担心，GitHub 非常负责，它像大多数公司那样，积极维护 API 并为其提供支持。不过要知道，API 中的 API 参考是值得信赖的，比外部文档可信，因为文档可能与 API 本身脱节。

API 中的映射富含数据。例如，映射不仅包含 URL，还有为 URL 提供参数的方式。在前面那个示例中，`code_search_url` 键对应的 URL 明显用于在 GitHub 中搜索代码，此外还指明了如何构建传给 URL 的参数。如果客户端够智能，能读懂这种纲领性的格式，就能动态生成查询，无需开发者去阅读 API 文档。至少，这是超媒体为我们指明的美好未来。如果你是怀疑论者，至少要知道 API（比如 GitHub）把文档嵌入自身当中，并且 GitHub 做了足够的测试，能够证明内嵌的文档与 API 端点传递的信息匹配。其他类型的 API 则没有这么强的保障。

下面，我们来简单讨论所有 GitHub API 的响应格式——JSON。

1.3 JSON格式

GitHub API 返回的所有响应都是 JSON（JavaScript Object Notation，JavaScript 对象表示法）格式。JSON 是一种“轻量级数据交换格式”（详情参见 JSON.org 网站：http://www.json.org/）。此外，还有其他与之相争且有效的格式，例如 XML（Extensible Markup Language，可扩展标记语言）和 YAML（YAML Ain’t Markup Language），不过 JSON 正在快速成为 Web 服务的事实标准。

JSON 之所以如此流行，有以下两个原因。

- JSON 易于阅读。与 XML 等序列化格式相比，JSON 很好地平衡了人类可读性。
- 只需小幅修改（和程序员的认知处理），JSON 就能在 JavaScript 中使用。在客户端和服务器端都能同样良好使用的数据格式一定会胜出，JSON 就是如此。

GitHub 最初使用 Ruby on Rails 工具栈开发（部分代码现在仍在运行），你可能觉得这样的网站应该支持指定替代格式（如 XML），但是 GitHub 不再支持 XML 了。JSON 万岁！

如果你用过其他基于文本的交换格式，会发现 JSON 特别简单。注意，JSON 只支持使用双引号，不支持单引号，这一点对于刚接触 JSON 的人来说可能有点难以理解，出乎意料。

我们使用 cURL 这个命令行工具从 GitHub API 中获取数据。如果再有一个简单的命令行工具能处理 JSON 就好了。下面介绍一个这样的工具。

1.3.1 在命令行中解析JSON

JSON 是一种文本格式，因此可以使用任何命令行文本处理工具处理 JSON 响应，例如令人敬仰的 AWK。有一个专门解析 JSON 的优秀工具能补足 cURL，值得一用，这便是 jq。通过管道（大多数 shell 使用“|”字符）把 JSON 内容传给 jq 后，可以使用过滤器轻易提取 JSON 片段。

安装 jq

jq 可以从源码安装，使用包管理器安装（例如 `brew` 或 `apt-get`），下载页面（http://stedolan.github.io/jq/download/）还有适用于 OS X、Linux、Windows 和 Solaris 的二进制文件。

下面深入前面的示例，从访问 api.github.com 后收到的 API 映射中提取感兴趣的信息：

```
$ curl https://api.github.com | jq '.current_user_url'
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100  2004  100  2004    0     0   4496      0 --:--:-- --:--:-- --:--:--  4493
"https://api.github.com/user"
```

发生了什么？jq 工具解析 JSON，然后使用 .current_user_url 过滤器从 JSON 响应中获取内容。再次查看响应，你会发现响应是个关联数组，里面有多个键值对。jq 使用那个关联数组中的 current_user_url 作为键获取对应的值。

你还会发现 cURL 把传输时间信息打印出来了。cURL 把这些信息打印到标准错误。这是 shell 的一种约定，用于输出错误。jq 会正确忽略这个输出流（也就是说，JSON 格式的数据流不会被错误消息搅乱）。如果想禁止那些信息，让请求清晰明了，可以使用 -s 开关，在“静默”模式中运行 cURL。

jq 在 JSON 响应上应用过滤器的方式易于理解。下面通过一个复杂的请求（例如，获取某个用户的所有公开仓库）说明如何使用 jq 的模式参数。我们要获取一组更为复杂的信息，即用户的仓库列表，以此说明如何使用 jq 从响应中提取信息：

```
$ curl -s https://api.github.com/users/xrd/repos
[
  {
    "id": 19551182,
    "name": "a-gollum-test",
    "full_name": "xrd/a-gollum-test",
    "owner": {
      "login": "xrd",
      "id": 17064,
      "avatar_url":
      "https://avatars.githubusercontent.com/u/17064?v=3",
     ...
  }
]
$ curl -s https://api.github.com/users/xrd/repos | jq '.[0].owner.id'
17064
```

这个响应的结构与之前不同，它不是一个关联数组，而是一个普通数组（有多个元素）。为了获取第一个元素，我们要指定数字索引，然后再通过键进入元素里的关联数组，从而获取所需的内容——属主的 ID。

jq 工具能很好地检查 JSON 的有效性。前面说过，JSON 的键值对只能使用双引号，不能使用单引号。可以使用 jq 验证 JSON 是否有效，以及是否满足这个要求：

```
$ echo '{ "a" : "b" }' | jq '.'
{
  "a": "b"
}
$ echo "{ 'no' : 'bueno' }" | jq "."
parse error: Invalid numeric literal at line 1, column 7
```

传给 jq 的第一个 JSON 是有效的，而第二个 JSON 使用了无效的单引号字符，因此报错了。jq 过滤器是以字符串形式传递的参数，而把字符串提供给 jq 的 shell 不管你使用的是单引号还是双引号，从上述代码可以看出这一点。如果你不知道 echo 命令的作用，我告诉

你，它会把传给它的任何字符串打印出来。把这个命令和管道符号结合起来，可以通过标准输入轻易地把字符串提供给 jq。

jq 是个强大的工具，能从任何 JSON 响应中迅速获取内容。此外，jq 还有很多强大的功能，详情参见文档（https://stedolan.github.io/jq/）。

至此，我们知道如何仅使用一行命令完成从 GitHub API 中获取所需的信息，并从响应中解析出信息片段。可是，有时为 cURL 或 API 指定的参数可能是错误的，获得的数据不是我们想要的。接下来，我们学习如何调试 cURL 工具和 API 服务本身，从而在出错时获取更多的信息。

1.3.2　cURL的调试开关

前面说过，cURL 是验证响应是否与预期相符的绝佳工具。响应主体很重要，但是通常还要获取首部。为 cURL 指定 `-i` 和 `-v` 开关能轻易获取这些信息。`-i` 开关打印请求首部，`-v` 开关则打印请求和响应首部（> 符号表示请求数据，< 符号表示响应数据）。

```
$ curl -i https://api.github.com
HTTP/1.1 200 OK
Server: GitHub.com
Date: Wed, 03 Jun 2015 19:39:03 GMT
Content-Type: application/json; charset=utf-8
Content-Length: 2004
Status: 200 OK
X-RateLimit-Limit: 60
...
{
  "current_user_url": "https://api.github.com/user",
  ...
}
$ curl -v https://api.github.com
* Rebuilt URL to: https://api.github.com/
* Hostname was NOT found in DNS cache
*   Trying 192.30.252.137...
* Connected to api.github.com (192.30.252.137) port 443 (#0)
* successfully set certificate verify locations:
*   CAfile: none
  CApath: /etc/ssl/certs
* SSLv3, TLS handshake, Client hello (1):
* SSLv3, TLS handshake, Server hello (2):
...
* CN=DigiCert SHA2 High Assurance Server CA
*        SSL certificate verify ok.
> GET / HTTP/1.1
> User-Agent: curl/7.35.0
> Host: api.github.com
> Accept: */*
>
< HTTP/1.1 200 OK
```

```
* Server GitHub.com is not blacklisted
...
```

指定 -v 开关能获取所有信息：DNS 查询、SSL 证书链中的信息，以及完整的请求和响应信息。

> 注意，如果打印首部，jq 这样的工具会迷惑不解，因为提供的不是纯粹的 JSON。

本节旨在说明不仅主体（JSON 数据）中有值得关注的信息，首部中也有。我们要知道有哪些首部，还要知道哪些是重要的。根据 HTTP 规范，需要的首部有很多，这些首部通常可以忽略，不过如果请求不是相互独立的，有些首部则是必不可少的。

1.4　重要的首部

GitHub API 的每个响应中都有三个用于指明 API 频率限制的首部，分别是 X-RateLimit-Limit、X-RateLimit-Remaining 和 X-RateLimit-Reset。这些限制在 1.7.6 节详述。

从 GitHub API 中获取文本或 blob 内容时，要用到 X-GitHub-Media-Type 首部中包含的信息。向 GitHub API 发起请求时，可以在请求中发送 Accept 首部，指明想使用的格式。

下面，我们使用一个响应构建另一个响应。

1.5　跟随超媒体API

我们要使用 API 基端点返回的“映射”，手动生成另一个请求：

```
$ curl -i https://api.github.com/
HTTP/1.1 200 OK
Server: GitHub.com
Date: Sat, 25 Apr 2015 05:36:16 GMT
...
{
  "current_user_url": "https://api.github.com/user",
  ...
  "organization_url": "https://api.github.com/orgs/{org}",
  ...
}
```

我们可以使用组织的 URL，把占位符替换成 "github"：

```
$ curl https://api.github.com/orgs/github
{
  "login": "github",
  "id": 9919,
  "url": "https://api.github.com/orgs/github",
  ...
  "description": "GitHub, the company.",
  "name": "GitHub",
  "company": null,
  "blog": "https://github.com/about",
  "location": "San Francisco, CA",
  "email": "support@github.com",
  ...
  "created_at": "2008-05-11T04:37:31Z",
  "updated_at": "2015-04-25T05:17:01Z",
  "type": "Organization"
}
```

通过上述信息可以得知 GitHub 的一些信息。我们得知 GitHub 的博客地址是 https://github.com/about，公司位于旧金山，组织的创建日期是 2008 年 5 月 11 日。从博客中一篇发布于 4 月份的文章（https://github.com/blog/40-we-launched）中得知，GitHub 公司在这一个月之前就成立了。或许公司成立一个月之后才在 GitHub 网站中添加组织功能吧。

目前，我们发起的请求都用于获取公开信息。GitHub API 提供的信息十分丰富，但是验证身份之后才能访问隐私信息和不可公开使用的服务。例如，想使用 API 向 GitHub 中写入数据的话，需要知道如何验证身份。

1.6 身份验证

向 GitHub API 发起请求时，有两种身份验证方式：用户名和密码（HTTP 基本验证）以及 OAuth 令牌。

1.6.1 用户名和密码验证

提供用户名和密码后可以访问 GitHub 中受保护的内容。用户名验证使用 HTTP 基本验证实现，在 cURL 中使用 `-u` 标志指定。HTTP 基本验证就是用户名和密码验证：

```
$ curl -u xrd https://api.github.com/rate_limit
Enter host password for user 'xrd': xxxxxxxx
{
  "rate": {
    "limit": 5000,
    "remaining": 4995,
    "reset": 1376251941
  }
}
```

上述 cURL 命令先通过 GitHub API 的身份验证，然后获取该用户账号具体的频率限制信息。这些是受保护的信息，只有登录的用户才能查看。

1. 用户名验证的优点

几乎所有客户端库都支持 HTTP 基本验证。我们即将介绍的 GitHub API 客户端都支持用户名和密码验证。而且，自己实现客户端也很容易，因为这是 HTTP 标准的核心功能。所以，开发客户端时只要使用了符合标准的 HTTP 库，就能访问 GitHub API 中的内容。

2. 用户名验证的缺点

使用用户名和密码验证管理 GitHub API 的访问权限不合适，原因有如下几个。

- HTTP 基本验证是旧协议，没有预料到 Web 服务的粒度如此细化。如果通过用户名和密码验证用户的身份，那么无法指定让 Web 服务开放哪些特定的功能。
- 如果使用用户名和密码在手机端访问 GitHub API 的内容，又在笔记本电脑中访问 API 的内容，那就无法禁止某一台设备访问，只能都禁止。
- HTTP 基本验证无法扩展验证流程。现今，很多现代服务都支持双重身份验证，为了把这种验证方式插入现有的验证过程，需要修改 HTTP 客户端（例如 Web 浏览器），至少也要修改客户端预期的流程（让浏览器重复请求）。

这些问题在 OAuth 流程中都能得到解决（或者至少得到支持）。鉴于这些缺点，仅当便利性至上时才应当使用用户名和密码验证身份。

1.6.2　OAuth

OAuth 是一种身份验证机制，把令牌与功能或客户端绑定起来。也就是说，可以指定让服务为 OAuth 令牌开放哪些功能；还可以颁发多个令牌，与不同的客户端绑定，例如手机应用、笔记本电脑、智能手表，甚至是接入物联网的烤箱。更重要的是，可以吊销令牌而不对其他令牌产生影响。

OAuth 令牌主要的缺点是增加了一层复杂度，如果你只用过 HTTP 基本验证，对此可能不熟悉。HTTP 基本验证通常只需在 HTTP 请求中添加一个额外的首部，或者在客户端工具（如 cURL）中添加一个额外的标志。

OAuth 能解决上述问题，方法是把令牌限定在作用域（指定 Web 服务的功能子集）中，以及按需为不同的客户端生成不同的令牌。

1. 作用域：指定验证令牌可执行的操作

生成 OAuth 令牌时，要指定所需的访问权限。下述示例虽然使用 HTTP 基本验证创建令牌，但是一旦得到令牌，后续请求就不再需要使用 HTTP 基本验证。获颁 OAuth 令牌之后，便有权限读写相应用户的公开仓库。

下述 cURL 命令使用 HTTP 基本验证请求令牌：

```
$ curl -u username -d '{"scopes":["public_repo"]}' \
https://api.github.com/authorizations
{
  "id": 1234567,
  "url": "https://api.github.com/authorizations/1234567",
  "app": {
    "name": "My app",
    "url": "https://developer.github.com/v3/oauth_authorizations/",
    "client_id": "00000000000000000000"
  },
  "token": "abcdef87654321
  ...
}
```

请求成功后获得的 JSON 响应中有个令牌，把它提取出来之后可以提供给应用，用于访问 GitHub API。

如果使用双重身份验证，这个过程需要额外的步骤，详情参阅第 8 章。

令牌的使用方法是，在 Authorization 首部中指定令牌：

```
$ curl -H "Authorization: token abcdef87654321" ...
```

作用域明确了服务或应用能如何使用 GitHub API 中的数据。对于供用户自己使用的令牌来说，作用域便于稽查用户如何使用 API 提供的信息。不过，当第三方应用想访问你的信息时，最能体现作用域明确访问权限的重要价值和防护作用，因为你能确保应用只能访问允许它访问的数据，而且便于取消访问权限。

2. 作用域的不足

要知道，作用域有个极大的不足之处：无法精细调整特定仓库的访问权限。如果允许访问任何一个私有仓库，那么所有仓库就都能访问。

未来，GitHub 可能会修改作用域的工作方式，解决其中一些问题。OAuth 机制的好处是，发生变化后，只需请求重新设定作用域的令牌，应用的身份验证流程则无需修改。

> 构建服务或应用时，要特别谨慎地对待请求的作用域。用户会担忧交给你的数据是否安全（这是正常的），他们会根据请求的作用域评估能否信任应用。如果用户觉得不需要那么广的作用域，请求授权时一定要从提供给 GitHub 的权限列表中删除，与用户建立一定的信任之后再考虑升级为更广的作用域。

3. 逐步升级作用域

你可以先请求严格受限的作用域，以后再请求更广的作用域。例如，用户首次访问你的应

用时，你可以只获取 user 作用域，在你的服务中创建用户对象，仅当应用需要获取用户的仓库信息时再请求升级权限。此时，用户需要接受或拒绝升级请求。但是，事无巨细，（在与用户建立关系之前）什么都询问，往往会导致用户放弃登录。

下面详述使用 OAuth 验证身份的细节。

4. 简化的OAuth流程

OAuth 有很多版本，GitHub 使用的是 OAuth2。OAuth2 验证身份的流程如下：

(1) 应用请求访问；
(2) 服务提供方（GitHub）请求验证身份，通常使用用户名和密码；
(3) 如果启用了双重身份验证，询问 OTP（one-time password，一次性密码）；
(4) GitHub 返回包含令牌的 JSON 响应；
(5) 应用使用 OAuth 令牌请求 API。

接下来说明 GitHub API 在通信过程中用于提供反馈的各个 HTTP 状态码。

1.7 状态码

GitHub API 使用 HTTP 状态码明确表明请求的处理结果。如果使用的是简单的客户端，如 cURL，在获取数据之前一定要先验证状态码。如果自己编写 API 客户端，首先要重点关注状态码。刚开始使用 GitHub API 的用户应该仔细检查响应的状态码，熟悉请求可能致错的各种状况。

1.7.1 成功（200或201）

只要你用过 HTTP 客户端，就知道 HTTP 状态码“200”表示成功。请求的目标地址和相关的参数正确时，GitHub 响应的状态码是 200。如果请求在服务器中创建内容，响应的状态码是 201，表示成功在服务器中创建了内容。

```
$ curl -s -i https://api.github.com | grep Status
Status: 200 OK
```

1.7.2 不合规的JSON（400）

如果载荷（请求中发送的 JSON）无效，GitHub API 的响应是 400 错误，如下所示：

```
$ curl -i -u xrd -d 'yaml: true' -X POST https://api.github.com/gists
Enter host password for user 'xrd':
HTTP/1.1 400 Bad Request
Server: GitHub.com
Date: Thu, 04 Jun 2015 20:33:49 GMT
Content-Type: application/json; charset=utf-8
```

```
Content-Length: 148
Status: 400 Bad Request
...

{
  "message": "Problems parsing JSON",
  "documentation_url":
  "https://developer.github.com/v3/oauth_authorizations/#create...authorization"
}
```

这里，我们打算使用 Gist API 文档（https://developer.github.com/v3/gists/#create-a-gist）中给出的端点新建一个 Gist。后面有一章会详细讨论 Gist。这个请求失败了，因为我们使用的不是 JSON（看起来使用的像是 YAML，参见第 6 章）。载荷使用 `-d` 开关发送。GitHub 在返回的 JSON 响应中使用 `documentation_url` 键提供了一个地址，告诉你在哪里寻找正确格式的文档。注意，我们使用了 `-X` 开关并把值设为 `POST`，这么做是为了告诉 cURL 向 GitHub 发起 POST 请求。

1.7.3　错误的JSON（422）

如果请求中有任何无效的字段，GitHub 会响应 422 错误。下面我们尝试修复前面那个请求。根据文档，JSON 载荷应该使用下述格式：

```
{
  "description": "the description for this gist",
  "public": true,
  "files": {
    "file1.txt": {
      "content": "String file contents"
    }
  }
}
```

如果 JSON 是有效的，但是字段不正确呢？

```
$ curl -i -u chris@burningon.com -d '{ "a" : "b" }' -X POST
https://api.github.com/gists
Enter host password for user 'chris@burningon.com':
HTTP/1.1 422 Unprocessable Entity
...

{
  "message": "Invalid request.\n\n\"files\" wasn't supplied.",
  "documentation_url": "https://developer.github.com/v3"
}
```

注意两件事。其一，返回的是 422 错误，表示 JSON 有效，但是字段不正确。其二，获得一个说明错误原因的响应：请求载荷中缺少 `files` 键。

1.7.4　成功创建（201）

我们已经知道 JSON 无效时会发生什么，那么如果请求发送的 JSON 是有效的呢？

```
$ curl -i -u xrd \
-d '{"description":"A","public":true,"files":{"a.txt":{"content":"B"}}} \
https://api.github.com/gists
Enter host password for user 'xrd':
HTTP/1.1 201 Created
...

{
  "url": "https://api.github.com/gists/4a86ed1ca6f289d0f6a4",
  "forks_url":
  "https://api.github.com/gists/4a86ed1ca6f289d0f6a4/forks",
  "commits_url":
  "https://api.github.com/gists/4a86ed1ca6f289d0f6a4/commits",
  "id": "4a86ed1ca6f289d0f6a4",
  "git_pull_url": "https://gist.github.com/4a86ed1ca6f289d0f6a4.git",
  ...
}
```

成功！我们创建了一个 Gist，而且获得了表示正确处理的 201 状态码。为了让命令更易于阅读，我们使用反斜线，把参数写成多行。此外，注意 JSON 不需要空格，因此我们把传给 `-d` 开关的字符串中的空格全都去掉了（为了节省空间，让命令更容易阅读一些）。

1.7.5　完全没变化（304）

304 与 200 的作用类似：告诉客户端请求成功。不过，304 多提供了一些信息，告诉客户端自上次请求以来数据没有变化。对于担心用量限制的用户来说（大多数用户都会担心），这是重要的信息。我们还未讲解频率限制的运作方式，所以下面先讨论这个话题，然后再演示如何使用条件首部触发 304 响应码。

1.7.6　GitHub API的频率限制

GitHub 会设法限制用户请求 API 的频率。匿名请求（没有使用用户名和密码或者 OAuth 令牌验证身份的请求）的限制为一小时 60 次。如果开发一个与 GitHub API 集成的系统，代表用户执行操作，一小时 60 次请求显然不够用。

验证身份后，向 GitHub API 发起请求的频率会增加到每小时 5000 次。虽然这比匿名请求的频率多了两个数量级，但是若想使用你自己的 GitHub 凭据代表很多用户发起请求，仍然有问题。

鉴于此，如果你的网站或服务使用 GitHub API 请求 GitHub API 中的信息，应该考虑使用 OAuth，并且使用用户共享的身份验证信息请求 GitHub API。如果使用其他用户的 GitHub

账户的令牌，那么频率限制算在该用户身上，而不是算在你的账户上。

> 其实，频率限制有两种：核心频率限制和搜索频率限制。前面几段说明的是核心频率限制。对搜索来说，验证身份的用户每分钟不能发起超过 20 个请求，匿名用户每分钟不能超过 5 个请求。这里假定搜索请求消耗的基础设施资源更多，因此对用量的限制更严格。

注意，GitHub 按 IP 地址记录匿名请求。因此，如果你身处防火墙后面，还有其他用户发起匿名请求，那么所有这些请求会算在一起。

1.7.7 获知频率限制

获知频率限制的方法很简单，向 `/rate_limit` 发起 GET 请求即可。返回的 JSON 文档中有要遵守的频率限制、剩余的请求数和时间戳（1970 年之后的秒数）。注意，时间戳的时区是 UTC（Coordinated Universal Time，协调世界时）。

下述命令清单使用 cURL 获取匿名请求的频率限制。为了节省空间，部分响应省略了，不过你能注意到配额信息出现了两次：一次在 HTTP 响应的首部中，一次在 JSON 响应中。每次请求 GitHub API 都会返回频率限制首部，因此不太有必要直接请求 `/rate_limit`：

```
$ curl https://api.github.com/rate_limit
{
  "resources": {
    "core": {
      "limit": 60,
      "remaining": 48,
      "reset": 1433398160
    },
    "search": {
      "limit": 10,
      "remaining": 10,
      "reset": 1433395543
    }
  },
  "rate": {
    "limit": 60,
    "remaining": 48,
    "reset": 1433398160
  }
}
```

一小时 60 次请求不是特别多，如果计划做些有趣的事，可能很快就会超出这一限制。如果每分钟 60 次请求的限制快到了，你可能要想办法验证身份，然后再请求 GitHub API。讨论身份验证请求时会说明做法。

对 `/rate_limit` 的请求算在频率限制内。记住，频率限制在 24 小时后重置。

1.8 使用条件请求规避频率限制

如果请求 GitHub API 的目的是获悉一个用户或一个仓库的活动数据，很有可能不会返回太多数据。如果每隔几分钟检查有没有新活动，在一段时间内可能没有任何活动。这种持久轮询虽然有时不传送新活动，但是仍然消耗着请求的频率限制。

此时，可以发送 `If-Modified-Since` 和 `If-None-Match` 两个 HTTP 条件首部，让 GitHub 返回表示什么都没变的 HTTP 304 响应码。如果发送的请求包含条件首部，而且 GitHub API 返回 HTTP 304 响应码，这样的请求不会从频率限制中扣除。

下述命令清单举例说明如何把 `If-Modified-Since` 首部传给 GitHub API。这里我们指明，仅当 Twitter Bootstrap 仓库在 2013 年 8 月 11 日（星期日）下午 7：49（GMT）之后有变化才接收内容。GitHub API 响应的状态码是 304，而且告诉我们，仓库的最后一次变动是在截止时间的前一分钟。

```
$ curl -i https://api.github.com/repos/twbs/bootstrap \
          -H "If-Modified-Since: Sun, 11 Aug 2013 19:48:59 GMT"
HTTP/1.1 304 Not Modified
Server: GitHub.com
Date: Sun, 11 Aug 2013 20:11:26 GMT
Status: 304 Not Modified
X-RateLimit-Limit: 60
X-RateLimit-Remaining: 46
X-RateLimit-Reset: 1376255215
Cache-Control: public, max-age=60, s-maxage=60
Last-Modified: Sun, 11 Aug 2013 19:48:39 GMT
```

GitHub API 也能理解 HTTP 缓存标签。ETag（Entity Tag）是一个 HTTP 首部，用于确定之前缓存的内容是否为最新版。系统可能会像下面这样使用 ETag。

- 客户端从 HTTP 服务器中请求信息。
- 服务器返回一个 ETag 首部，标记内容的一个版本。
- 客户端在后续的所有请求中发送这个 ETag 首部：
  - 如果服务器有较新的版本，返回新内容和新 ETag；
  - 如果服务器没有较新的版本，返回 HTTP 304 响应码。

下述命令清单演示两个命令。第一个 cURL 命令访问 GitHub API，生成 ETag 值；第二个命令在 `If-None-Match` 首部中传送那个 ETag 值。你能注意到，第二个响应是 HTTP 304，即告知调用方没有新内容。

```
$ curl -i https://api.github.com/repos/twbs/bootstrap
HTTP/1.1 200 OK
Cache-Control: public, max-age=60, s-maxage=60
Last-Modified: Sun, 11 Aug 2013 20:25:37 GMT
ETag: "462c74009317cf64560b8e395b9d0cdd"
```

```
{
  "id": 2126244,
  "name": "bootstrap",
  "full_name": "twbs/bootstrap",
  ....
}

$ curl -i https://api.github.com/repos/twbs/bootstrap \
        -H 'If-None-Match: "462c74009317cf64560b8e395b9d0cdd"'

HTTP/1.1 304 Not Modified
Status: 304 Not Modified
Cache-Control: public, max-age=60, s-maxage=60
Last-Modified: Sun, 11 Aug 2013 20:25:37 GMT
ETag: "462c74009317cf64560b8e395b9d0cdd"
```

建议你使用条件请求首部，这样能节省资源，还能确保支持 GitHub API 的基础设施不生成不必要的内容。

目前，我们都使用 cURL 客户端访问 GitHub API，只要网络允许，我们可以做任何想做的事。此外，GitHub API 还能使用其他方式访问，例如使用浏览器，不过此时有些特定的限制，讨论如下。

1.9 在Web中访问内容

如果使用服务器端程序或者命令行访问 GitHub API，只要网络允许，可以发起任何请求。如果想在浏览器中使用 JavaScript 和 XHR（XmlHttpRequest）对象访问 GitHub API，要知道浏览器同源策略施加的限制。简单来说，在 JavaScript 中，不能使用标准的 XHR 请求访问源页面所在域名之外的域名。若想绕开这个限制，有两个选择，一个很巧妙（JSON-P），另一个支持全面，但是稍微麻烦点（CORS）。

1.9.1 JSON-P

JSON-P 算是一种浏览器 hack，目的是避开同源策略，获取其他服务器中的信息。JSON-P 之所以可用，是因为同源策略不检查 `<script>` 标签；也就是说，页面可以从源服务器之外的服务器中引用内容。JSON-P 的用法是，在 JavaScript 文件中使用特殊的方式编码载荷数据，将其载入自己实现的回调函数中处理。GitHub API 支持这种句法：请求脚本时为 URL 提供一个参数，指明加载完脚本后执行哪个回调。

在 cURL 中可以模拟这样的请求：

```
$ curl https://api.github.com/?callback=myCallback
/**/myCallback({
  "meta": {
```

```
    "X-RateLimit-Limit": "60",
    "X-RateLimit-Remaining": "52",
    "X-RateLimit-Reset": "1433461950",
    "Cache-Control": "public, max-age=60, s-maxage=60",
    "Vary": "Accept",
    "ETag": "\"a5c656a9399ccd6b44e2f9a4291c8289\"",
    "X-GitHub-Media-Type": "github.v3",
    "status": 200
  },
  "data": {
    "current_user_url": "https://api.github.com/user",
    "current_user_authorizations_html_url":
    "https://github.com/settings/connections/applications{/client_id}",
    "authorizations_url": "https://api.github.com/authorizations",
    ...
  }
})
```

如果在网页中的 `<script>` 标签里使用上述代码中的 URL（`<script src="https://api.github.com/?callback=myCallback" type= "text/javascript"></script>`），浏览器会加载上述代码中的内容，把数据传给你定义的 `myCallback` 回调函数执行。在网页中可以像下面这样实现回调函数：

```
<script>
function myCallback( payload ) {
  if( 200 == payload.status ) {
    document.getElementById("success").innerHTML =
      payload.data.current_user_url;
  } else {
    document.getElementById("error").innerHTML =
      "An error occurred";
  }
}
</script>
```

这个示例演示如何从载荷数据中获取 `current_user_url`，把它放入一个 div 元素，例如 `<div id="success"> </div>`。

JSON-P 通过 `<script>` 标签实现，因此只支持向 API 发起 GET 请求。如果只需要 API 的只读权限，多数情况下 JSON-P 能满足需求，而且易于配置。

如果你觉得 JSON-P 局限太多或者不优雅，可以使用 CORS。这是在网页中访问外部服务的官方方式，不过较复杂。

1.9.2 CORS支持

CORS 用于从源主机之外的域名中访问内容，这是 W3C（一个 Web 标准组织）认可的方式。CORS 要求事先正确配置服务器，查询时必须表明自己允许跨域请求。如果服

务器明确表示，“是的，你可以从其他域名中访问我的内容”，那么就允许 CORS 请求。HTML5Rocks 网站中有篇优秀的教程（http://www.html5rocks.com/en/tutorials/cors/），解说了 CORS 的诸多细节。

因为使用 CORS 的 XHR 支持从同一域名获取的同类 XHR 请求，所以除了 GET 请求之外，还能向 GitHub API 发起 POST、DELETE 和 UPDATE 请求。JSON-P 和 CORS 是在 Web 浏览器中访问 GitHub API 的两种方式，前者简单，后者强大，但是需要额外配置。

我们可以使用 cURL 证明 GitHub API 能正确响应 CORS 请求。这里我们只关注首部，因此要使用 `-I` 开关，让 cURL 发起 HEAD 请求，告诉服务器别响应主体内容。

```
curl -I https://api.github.com
HTTP/1.1 200 OK
Server: GitHub.com
...
X-Frame-Options: deny
Content-Security-Policy: default-src 'none'
Access-Control-Allow-Credentials: true
Access-Control-Expose-Headers: ETag, Link, X-GitHub-OTP,
X-RateLimit-Limit, X-RateLimit-Remaining, X-RateLimit-Reset,
X-OAuth-Scopes, X-Accepted-OAuth-Scopes, X-Poll-Interval
Access-Control-Allow-Origin: *
X-GitHub-Request-Id: C0F1CF9E:07AD:3C493B:557107C7
Strict-Transport-Security: max-age=31536000; includeSubdomains;
preload
```

可以看到，`Access-Control-Allow-Credentials` 首部的值是 `true`。有些 JavaScript 宿主浏览器会自动发起预检（preflight）请求，验证这个首部的值是否为 `true`（此外还会验证 `Access-Control-Allow-Origin` 等其他首部是否正确设置，以便允许处理来自相应域的请求）；有些浏览器则要求我们自己动手发起预检请求。自动预检还是手动预检由浏览器的实现决定。浏览器通过这些首部确认支持 CORS 后，可以向 GitHub API 所在的域发起 XHR 请求，这与向相同的域发起的任何其他 XHR 请求没有差别。

我们大致讲解了如何连接 GitHub API，以及 GitHub API 的细节。此外，GitHub API 还有一些其他用途，例如使用这项服务按需渲染内容。

1.9.3　指定响应的内容格式

向 GitHub API 发送请求时可以指定期望的响应格式。比方说，如果请求的内容包含提交评论中的文本，可以使用 `Accept` 首部指定获取原始的 Markdown 还是 Markdown 生成的 HTML。此外，还可以指定使用 GitHub API 的哪个版本。当下，可以指定使用 API 的第 3 版或 beta 版。

获取格式化内容

随请求发送的 `Accept` 首部可以影响 GitHub API 返回的文本格式。下面举个例子。假设你想读取一个 GitHub 工单的正文。工单的正文以 Markdown 格式存储，默认作为请求的响应返回。如果想使用 HTML 渲染响应而不是 Markdown，可以发不同的 `Accept` 首部，如下述 cURL 命令所示：

```
$ URL='https://api.github.com/repos/rails/rails/issues/11819'
$ curl -s $URL | jq '.body'
"Hi, \r\n\r\nI have a problem with strong...." ❶
$ curl -s $URL | jq '.body_html'
null ❷
$ curl -s $URL \
-H "Accept: application/vnd.github.html+json" | jq '.body_html'
"<p>Hi, </p>\n\n<p>I have a problem with..." ❸
```

❶ 没有指定额外的首部，获取数据的内在表述，即 Markdown。

❷ 注意，如果不请求 HTML 表示，默认情况下 JSON 响应中就没有 HTML 格式内容。

❸ 如果像第三个命令那样指定 `Accept` 首部，JSON 响应中会包含渲染成 HTML 的正文。

除了"raw"和"html"之外，还有两个格式选项会影响 GitHub API 分发 Markdown 内容的方式。如果把格式指定为"text"，工单的正文会以纯文本格式返回。如果指定为"full"，内容会经过多次渲染，包括原始的 Markdown、渲染后的 HTML 和渲染后的纯文本。

除了控制文本内容的格式之外，获取 blob 时还可以指定返回原始的二进制文件还是 base64 编码的文本。获取提交时，还可以指定返回内容的 diff 格式还是 patch 格式。精确控制格式的详细信息参见 GitHub API 的文档。

> GitHub 团队为 API 提供了非常详细的文档，包含使用 cURL 的示例。建议收藏这个 URL：https://developer.github.com/v3/，你经常会用到。注意，这个 URL 显然是针对当前的第 3 版 API，因此有新版发布的话，URL 会变。

1.10 小结

我们在本章学习了如何使用最简单的客户端（即命令行 HTTP 工具 cURL）访问 GitHub API。此外，我们通过分析 JSON 响应，探索了 GitHub API，还说明了如何结合使用 cURL 和命令行工具（jq），在往往包含大量数据的 GitHub API 响应中快速查找信息。我们学习了 GitHub 支持的不同身份验证机制，还学习了在浏览器中访问 GitHub API 的可行性和折中方案。

下一章说明 Gist 和 Gist API。我们将使用 Ruby 构建一个显示 Gist 的程序，把应用的所有源码文件都放在 Gist 中。

第 2 章

Gist和Gist API

GitHub 满足了程序员分享信息的热切期盼，为软件开发带来了变革。但是 GitHub 不仅仅是个“分享”工具，而且是众多工具的集合，这些工具去除了沟通障碍，简化了工作流程。这些工具的出现恰逢信息技术发生变革之时，在新形式下公司越来越多地采用助力远程工作的开放技术。

Gist 服务部分满足了这种需求。使用 Gist 可以私下分享和重用代码以及重构代码，还能以之前的重量级工具不能提供的方式进行试验。本章探讨如何使用 Gist 分享代码，然后使用 Gist API 构建一个托管在 Gist 中的应用。

2.1 简便的代码分享工具

创建 Gist 的方法很简单，在页面中部那个大大的文本框中粘贴代码片段，选择性输入描述或文件名，然后选择创建公开或秘密 Gist 即可。Gist 创建好之后会给你提供一个 URL，用于分享。大多数情况下，Gist 能自动检测代码的语言，然后根据语言高亮显示句法，如图 2-1 所示。

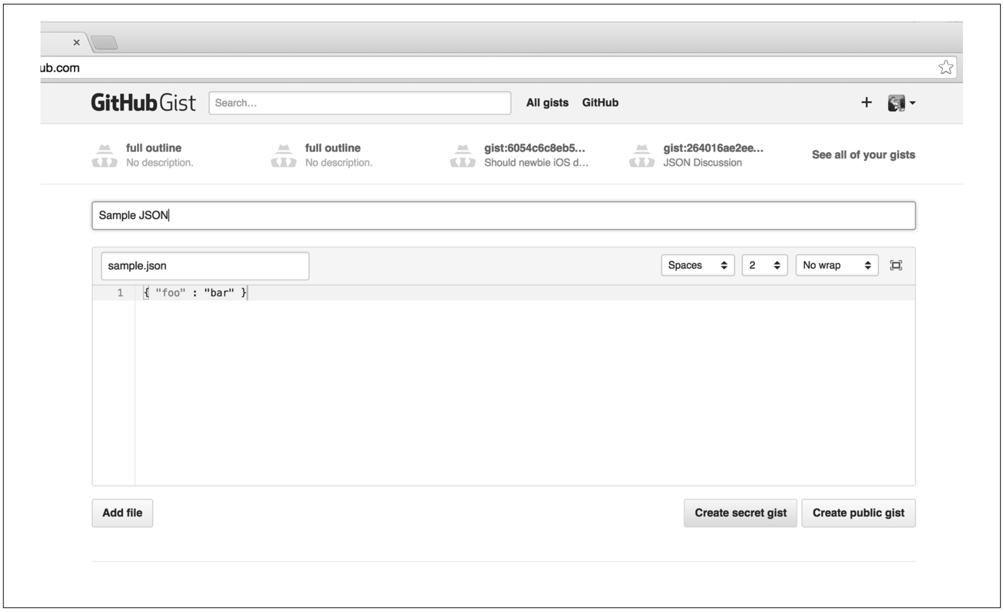


图 2-1：在 Gist 中编写 JSON

其他服务也能做到这一点：首先是 Pastebin，还有其他众多提供差异化代码分享服务的网站。可是，GitHub 提供的 Gist 不仅仅是一项代码粘贴服务。Gist 是强大的仓库，可派生，可编辑，功能全面。接下来我们将介绍一些基础知识，说明 Gist 是什么，如何创建 Gist，然后指出，除了分享代码之外，Gist 还是一个实实在在的应用。

2.2 Gist是仓库

每个 Gist 都是微型仓库，可以更新，可以使用 `git log` 命令查看历史，可以下载、修改之后再使用 `git push` 命令推送到 gist.github.com 中的仓库（这会在该公共网页中重新发布 Gist）。此外，Gist 与常规的仓库一样，可以派生。

在 Gist 仓库中可以创建分支，不过 gist.github.com/ 中不会显示分支。尽管如此，如果需要使用分支，仍然可以像往常一样在仓库中创建分支。推送之后，分支信息会保存在上游仓库中。

公开和秘密 Gist 的数量不限。多数情况下，可以用秘密 Gist 代替私有仓库，而且秘密 Gist 不算在 GitHub 付费账户对私有仓库的数量限制内。你也可以将 Gist 公开，然后把 URL 分享到邮件列表或你想得到公众反馈的其他地方。

Gist 分为两种（公开的和秘密的），理解二者之间的区别很重要。公开 Gist 能搜索到。秘密 Gist 搜索不到，但是只要知道 URL 就能访问。不要在 Gist 中发布任何机密代码，因为一旦发布，只有 URL 不为人知，代码才安全。

大多数人使用 URL 分享 Gist，不过 Gist 也可以嵌入其他页面（如博客），以简单而精美的方式显示代码片段。

2.2.1 在HTML中嵌入Gist

如果想把 Gist 嵌入 HTML 页面，在 Gist 左边寻找“Embed”框，复制框里的代码（如 <script src="https://gist.github.com/xrd/8923697.js"></script>），然后将其粘贴到 HTML 中。

如果只想嵌入 Gist 中某个特定的文件（前提是 Gist 包含多个文件），在 src 属性中 URL 的末尾加上 ?file=hi.rb。

2.2.2 在Jekyll博客中嵌入Gist

在 Jekyll 博客（参见第 6 章）中可以使用一个特殊的句法轻易嵌入 Gist。{% gist 8138797 %} 这个简洁的句法用于嵌入公开 Gist，地址是 http://gist.github.com/8138797。如果想使用 Gist 中的某个文件，在 Gist 编号后添加一个文件名，如 {% gist 8138797 hi.rb %}。秘密 Gist 也能嵌入，此时要在前面加上 Gist 所属账户的用户名，例如 {% gist xrd/8138797 hi.rb %}。

接下来说明脱离 GitHub.com 网站，如何使用命令行创建 Gist。

2.3 使用命令行创建Gist

执行 gem install gist 命令，安装一个命令行工具，用于协助创建 Gist。这个工具的用法简单，输入命令，然后输入想发布为 Gist 的数据即可：

```
$ gist
(type a gist. <ctrl-c> to cancel, <ctrl-d> when done)
{ "foo" : "bar" }
https://gist.github.com/9106765
```

gist 命令会返回刚创建的 Gist 的链接。默认，Gist 以匿名方式创建。如果想登录，要使用 --login 开关。登录后，创建的 Gist 会与账户关联起来：

```
$ gist --login
Obtaining OAuth2 access_token from github.
GitHub username: xrd
```

```
GitHub password:
2-factor auth code: 787878

Success! https://github.com/settings/applications
```

我们可以通过管道把文件的内容传给 `gist` 命令：

```
$ echo '{ "foo" : "bar" }' | gist
https://gist.github.com/9106799
```

还可以使用 `cat` 命令把文件传给 `gist` 命令：

```
$ cat MyJavaFile.java | gist
https://gist.github.com/9345609
```

Gist 通常用于显示好玩或出错的代码，有时你并不想显示文件的全部内容。此时，可以使用命令行工具 `grep`。`grep` 能搜索特定的代码段，与正确的开关结合后能把代码段和前后几行代码发布到 Gist 中。下述命令在 MyJavaFile.java 文件中搜索 `myFunction` 函数，然后获取接下来的 20 行代码，将其存入一个 Gist：

```
$ grep -A 20 myFunction MyJavaFile.java | gist
https://gist.github.com/9453069
```

添加 `-o` 开关后，会在默认的 Web 浏览器中打开 Gist。此外，可以使用 `-c` 开关把 Gist 的 URL 复制到剪贴板，或者使用 `-P` 开关把剪贴板中的内容复制到 Gist 里。

`gist` 命令还有很多其他有趣的功能，若想深入了解，执行 `gist` 命令时指定 `--help` 开关。

Gist 本身是仓库，因此有两种用途：一是托管代码示例；二是作为仓库，在里面存储功能完整的示例应用。

2.4　Gist是功能完整的应用

下面我们构建一个简单的 Sinatra 应用，以此说明托管在 Gist 中的代码也是真实可用的应用。Sinatra 是一个 Ruby 库，用于创建特别简单的 Web 服务器。Sinatra 程序可以像下面这样简单：

```
require 'sinatra'

get '/hi' do
  "Hello World!"
end
```

访问 gist.github.com，创建一个 Gist。正确输入上述代码，然后选择公开 Gist。

现在，我们创建了一个便于分享的 Gist，任何人都能查看里面的代码。更重要的是，这是

一个存有可执行代码的仓库。如果想克隆这个 Gist，在右边寻找 Clone URL。你应该会看到一个 Git 协议的 URL 和一个 HTTPS 协议的 URL。如果克隆的目的只是读取 Gist，可以使用 HTTPS 协议的 URL。严格来说，使用 HTTPS 协议的 URL 克隆仓库后，可以推送改动，不过启用双重身份验证后则不行。大多数情况下，使用 Git 协议的 URL 更简单也更灵活。

我们来克隆这个 Gist：

```
$ git clone git@gist.github.com:8138797.git
```

克隆完成之后，进入仓库。你会看到一组文件，目前这个仓库中只有一个文件：

```
$ cd 8138797
$ ls
hi.rb
```

这个代码可以执行，运行的方式是执行 `ruby hi.rb` 命令。

如果以前没用过 Sinatra，执行上述命令后会报错。这个程序需要一个名为“sinatra”的库，而我们还没安装那个库。我们可以编写一个自述文件，或者在那个文件中添加文档。保证用户正确安装所需库的另一个方法是使用 Gemfile 文件。这个文件用于指定需要安装哪些库，以及从哪里安装。听起来这是最好的方式。

```
$ printf "source 'https://rubygems.org'\ngem 'sinatra'" > Gemfile
```

`bundle` 命令（由 bundler gem 提供）会安装 Sinatra 及相关的依赖：

```
$ bundle
Using rack (1.5.2)
Using rack-protection (1.5.1)
Using tilt (1.4.1)
Using sinatra (1.4.4)
Using bundler (1.3.5)
Your bundle is complete!
Use `bundle show [gemname]` to see where a bundled gem is installed.
```

为什么要这么做呢？因为现在我们可以在本地仓库中添加 Gemfile 文件，然后发布到 Gist 中，在网上分享。现在，我们的仓库中不仅有代码，还有为人熟知的清单文件，指明运行代码所需的组件。

2.5 渲染Gist的Gist

下面把 Octokit 这个 Ruby gem 添加到应用中，用它拉取指定用户的所有公开 Gist。Octokit 是访问 GitHub API 的官方 Ruby 库。为什么要创建一个显示其他 Gist 的 Gist 呢？如今流行

自指的元代码，这是现代人对 René Magritte 的著名画作《这不是一个烟斗》的回应。[1]

在根目录中添加一个视图文件 index.erb：

```
<html>
<body>

User has <%= count %> public gists

</body>
</html>
```

在 Gemfile 文件中添加 Octokit gem：

```
gem "octokit"
```

运行 `bundle` 安装 Octokit。然后，把 hi.rb 应用改成下面这样：

```
require 'sinatra'
require 'octokit'

set :views, "."

get '/:username' do |username|
  user = Octokit.user username
  count = user.public_gists
  erb :index, locals: { :count => count }
end
```

现在，文件系统应该是下面这样，有三个文件：

```
$ ls -1
Gemfile
hi.rb
index.erb
```

按 Ctrl-C 键，然后执行 `ruby hi.rb`，重启 Sinatra。在浏览器中访问 http://localhost:4567/xrd，你会看到 `xrd` 用户公开 Gist 的数量（见图 2-2）。修改 URL 中的用户名，指定其他 GitHub 账户的用户名，你会看到页面中显示该用户最近发布的五个 Gist。

注 1：Ben Zimmer 对此解说得最好（http://www.bostonglobe.com/ideas/2012/05/05/dude-this-headline-meta-dude-this-headline-meta/it75G5CSqi82NtoQHIucEP/story.html?camp=pm）。



图 2-2：显示 Gist 的数量

2.5.1　深入了解Gist API

GitHub API 使用超媒体，而不是由资源驱动的简单 API。Octokit 这样的客户端把超媒体细节隐藏到优雅的 Ruby 代码之后了。不过，理解超媒体的工作方式仍有益处，这样便于深入 GitHub API，获取更多的信息。

大多数 REST 式 API 都有“网站地图”，这通常是一份 API 参考文档，告诉用户该用哪些端点。知道 API 提供了哪些资源之后，用户使用一些 HTTP 动词处理资源。超媒体看待 API 的方式有所不同。超媒体 API 在响应中通过“功能可见性”自我描述。这句话是什么意思呢？我们来看下面这个 API 的响应。

```
{
    "_links": {
        "self": {
            "href": "http://shop.oreilly.com/product/0636920030300.do"
        }
    }
    "id": "xrd",
    "name": "Chris Dawson"
}
```

可以看出，这个载荷中有一个 ID（`"xrd"`）和一个名字（`"Chris Dawson"`）。这个载荷根据 HAL Primer 文档（https://phlyrestfully.readthedocs.org/en/latest/halprimer.html）编写，那篇文档中有相关概念更详细的说明。

有一点要注意：超媒体 API 的载荷中包含数据本身的元数据，以及可能对数据执行的操作的元数据。REST 式 API 往往在载荷之外提供映射，因此使用这种 API 时，要使用特别的方式把 API 网站地图与数据结合起来。而对超媒体 API 来说，客户端能以正确且智能的方式处理载荷，完全用不到人类可读的文档中存储的网站地图。

这种松耦合让 API 和客户端具有灵活性。理论上，超媒体 API 与理解超媒体的客户端工作起来灵犀相通。修改 API 后，由于理解超媒体，客户端能发现变化，继续按预期运作。如

果使用 REST 式 API 的话，必须更新客户端（必须安装新版客户端），或者必须升级客户端代码。超媒体 API 的后端可能发生改动，此时，只要客户端能理解超媒体，就能自动且动态地确定从响应中获取信息的正确方式。也就是说，对超媒体客户端来说，API 后端变化时客户端代码不需要变。

《使用 HTML5 和 Node 构建超媒体 API》一书对这个话题做了深入阐述。

2.5.2　使用Octokit获取超媒体数据

现在我们稍微了解了超媒体，下面使用 Octokit 从中获取一些数据。

- 先在代码中创建一个资源，例如 `user = Octokit.user "xrd"`，以此初始化客户端。
- 现在 `user` 是一个对象，含有资源的真实数据。在这个示例中，可以调用 `user.followers` 方法查看我寥寥无几的关注者数量。
- `user` 对象中还有超媒体引用，调用 `user.rels` 方法可以看到这些引用。这个方法获取的是超媒体链接中描述的关系。
- 关系（通过调用 `user.rels` 方法获取）包括头像、自述、关注者，等等。
- 在某个关系上调用 `get.data` 方法可以访问并获取 GitHub API 中的数据（`followers = user.rels[:followers].get.data`）。
- 调用 `.get.data` 方法得到的是一个由关注者组成的数组（超过 100 个元素时分页）。

下面我们扩充这个 Sinatra 应用，使用超媒体引用获取关于用户的 Gist 的具体数据。

```
require 'sinatra'
require 'octokit'

set :views, "."

helpers do
  def h(text)
    Rack::Utils.escape_html(text)
  end
end

get '/:username' do |username|
  gists = Octokit.gists username, :per_page => 5
  erb :index, locals: { :gists => gists, username: username }
end
```

index.erb 文件中的代码用于迭代各个 Gist，获取 Gist 的内容。可以看出，响应对象是一个由 Gist 组成的数组，每个元素都有一个名为 `fields` 的属性。这个属性的值是各个 Gist 中所含文件的文件名。访问文件数组中的文件名会得到一个超媒体属性 `refs`。然后使用 Octokit 提供的 `.get.data` 方法获取原始内容（`raw`）：

```
<html>
<body>

<h2>User <%= username %>'s last five gists</h2>

<% gists.each do |g| %>
<% g[:files].fields.each do |f| %>
<b><%= f %></b>:

<%= h g[:files][f.to_sym].rels[:raw].get.data %>

<br/>
<br/>

<% end %>
<% end %>

</body>
</html>
```

现在，我们能看到 Gist 及其内容，如图 2-3 所示。

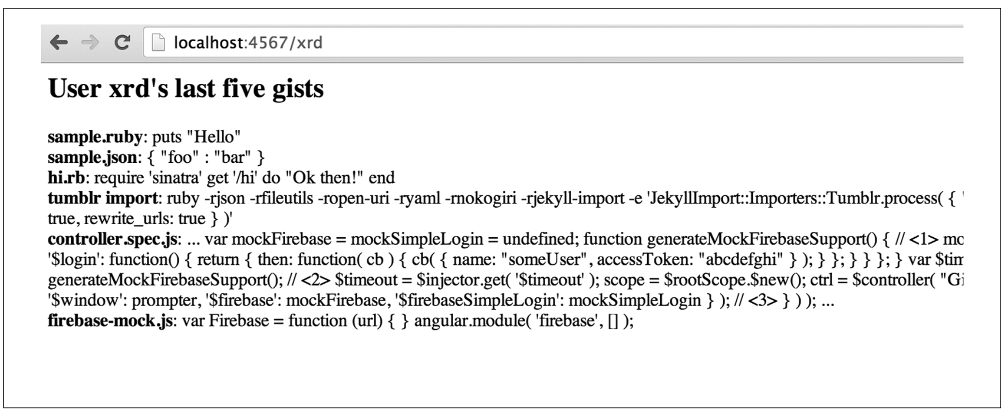


图 2-3：最新发布的五个 Gist，带有详细信息

2.6　小结

本章介绍了 Gist，学习了如何使用 Gist 分享代码片段。我们构建了一个简单的应用，将其存储为一个 Gist。这个应用首次使用高级语言编写的客户端库（使用 Ruby 编写的 Octokit 库）从 GitHub API 中获取数据。我们还深入了解了超媒体的运作方式，以及如何使用超媒体元数据实现一个客户端库。

下一章说明 GitHub 使用的维基库 Gollum，还会介绍用于访问 Git 仓库的 Ruby 库 Rugged 和用于访问 GitHub 的 Ruby 库。

第3章

GitHub使用的维基库Gollum

维基改变了我们创建和消化信息的方式。维基对技术项目（代码仓库）是个很好的补充，因为不会添加代码的非技术人员也能作出贡献。Gollum 是 GitHub 开发的开源维基库。Git 改变了协作编辑代码的方式，Gollum 则把 Git 的优势引入广泛使用的维基发布流程中。Gollum 维基本身是仓库，一般用于注解通常以代码为中心的仓库。通过 GitHub 的努力，任何仓库都能轻易关联一个维基。

本章探讨 Gollum 的基本用法。我们将在 GitHub 中创建一个维基，然后学习如何在 GitHub 中编辑它，以及如何在本地设备中将其作为一个仓库使用。接着，我们将自己动手在命令行中创建一个 Gollum 维基，指出至少要有哪些文件才能算是一个 Gollum 仓库。最后，我们将构建一个简单的图像整理工具，以一种完全不同的方式编辑 Gollum 维基，不过即便如此，仍能像常规的 Gollum 维基那样把信息发布到 GitHub 中。在这个过程中，我们会稍微探讨一下 Git 的内部机制。

> 本章会演示如何使用编程的方式修改 Git 仓库，无需深入理解 Git 的内部机理也能跟上节奏。《Git 版本控制管理》一书对本章（以及后面几章）是很好的补充。

3.1 “史麦戈的故事……”

最简单的 Gollum 维基是只有一个文件的 Git 仓库，这个文件是 Home.ext（ext 可以是任何被支持的维基标记格式，下文讨论）。

3.1.1 与仓库关联的维基

GitHub 中的任何仓库，不管是公开的还是私有的，都可以关联一个 Gollum 维基。如果想创建与仓库关联的维基，访问仓库页面，然后看最右边一栏。你会看到一个像书的图标，后面跟着 Wiki 这个单词，如图 3-1 所示。[1]

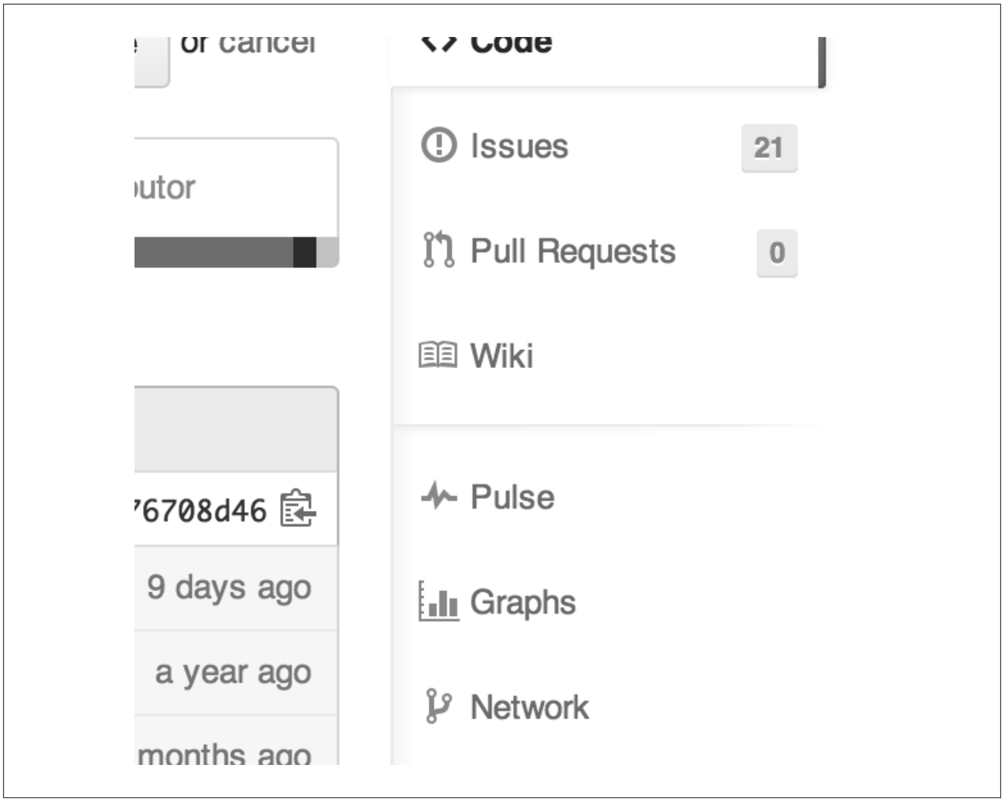


图 3-1：在侧边栏中访问关联的维基

首次点击那个链接后会把你带到一个页面，让你创建维基。GitHub 会要求你创建“首页”，即 Gollum 维基的入口（见图 3-2）。GitHub 会自动创建一个页面模板，以项目名称命名；你可以按照自己的需求修改模板。点击“Save Page”保存第一个页面，创建维基。

注 1：GitHub 的界面改版了，在新版中，维基的链接位于仓库名称下面。——译者注

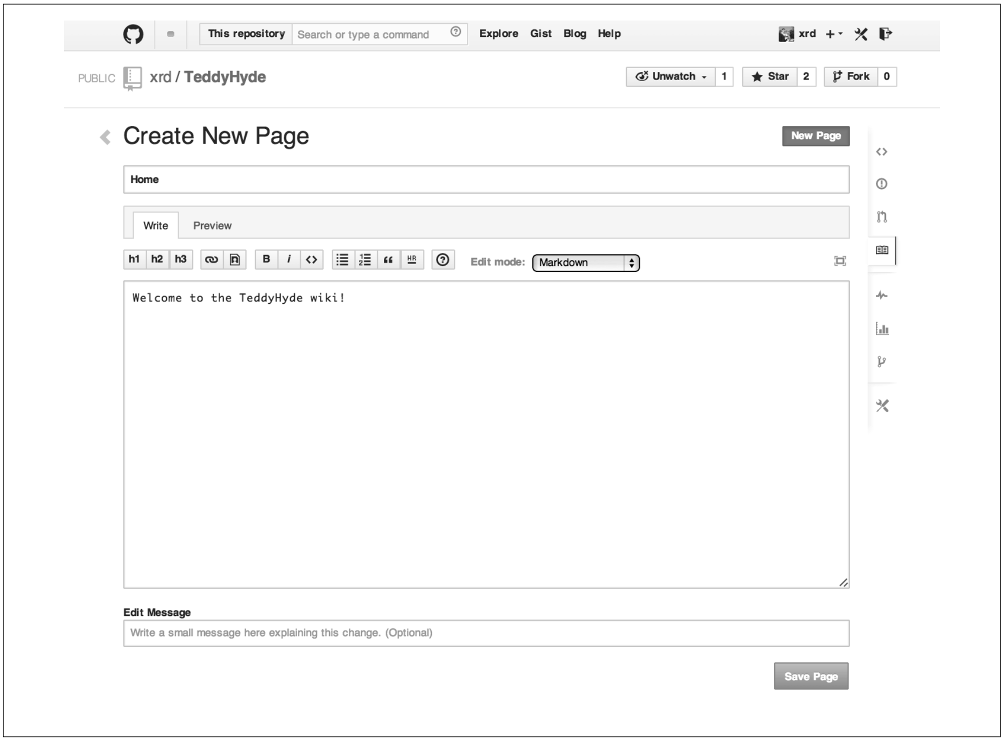


图 3-2：创建首页，新建一个维基

如果仓库是公开的，维基也是。公开仓库的维基是公开的，任何人都能访问。私有仓库的维基是私有的，只有有权限编辑仓库数据的用户或组织能访问。

下面说明 Gollum 维基支持的标记。

3.1.2　标记和结构

Gollum 中的文件可以使用 GitHub 支持的任何标记格式编写，包括 ASCIIdoc、Creole、Markdown、Org Mode、Pod、RDoc、ReStructuredText、Textile 和 MediaWiki。支持的标记语言多，灵活性高，却让人难以选择。Markdown（及其变体）是 GitHub 中最受欢迎的标记语言，在其他网站（如 Stack Overflow）中也广受欢迎。如果你不确定该使用哪种语言，使用 Markdown 最保险，因为它在 GitHub 中普遍使用。第 6 章会更为深入地介绍 Markdown。

对于选择使用 Markdown 的你来说，要知道除了 Markdown 标准的常规标签之外，Gollum 还添加了维基专用的标签。这些额外添加的标签大都与其他维基标记语言有细微区别（有的还不兼容），所以一定要阅读 Gollum 的文档（https://github.com/gollum/gollum/wiki）。下面说明其中几个最重要的标签。

1. 链接

显然，链接生成的是 HTML 中的 `<a>` 标签。不同标记格式使用的链接格式不同，Markdown 使用 `[text](URL)`，Gollum 添加了专用的链接标签：`[[Link]]`。

此外，还有如下几点需要注意。

- 可以使用竖线符号为链接添加标题：`[[http://foobar.com|A link to foobar]]`。
- 可以添加外部链接，也可以添加内部链接。
- `[[Review Images]]` 这样的链接会转换成链接到 review-images.ext（.ext 是你为维基选择的文件扩展名，极有可能是 Markdown）页面的相对链接。

维基通常由大量相互链接的页面组成，这种链接结构有利于编写页面。

> 前文说过，Gollum 维基与其他维基使用的标签虽然句法相似，但是仍有区别。例如，在 MediaWiki 中，带标题的链接的结构是相反的，如 `[[A link to foobar|http://foobar.com]]`，因此要注意。

2. 代码片段

Gollum 维基由 GitHub 开发，这是一个致力于改善软件开发者生活的公司，因此 Gollum 维基显然支持插入代码片段。插入代码片段的方法是，输入三个反引号，后面跟着可选的语言名，最后使用三个反引号结束代码块。如果指定语言名，Gollum 会正确高亮显示大部分语言的句法。

````
```ruby
def hello
  puts "hello"
end
```
````

以前，Gollum 支持使用下述句法引入任何 GitHub 仓库（以及任何分支）里的文件（https://github.com/gollum/gollum/wiki/Home/b16ce34e46e26600dc77bdf9b5aa6efcaf42026d#github-syntax-highlighting）：

````
```ruby:github:xrd/TeddyHyde/blob/master/Gemfile```
````

不过，现在已经不支持了。根据 Gollum 目前的文档，这个标签支持引入父级仓库里的文件：

````
```ruby:/lib/gollum/app.rb```
````

可是，我发现这样也不行。截至写作本书时，好像还没有办法把父级仓库（或任何其他仓库）里的代码插入维基的内容里，真是悲剧。

3. 结构组件

Gollum 支持添加侧边栏、页头和页脚。如果仓库中有名为 _Sidebar.ext 的文件，渲染后每个文件都会有一个侧边栏。所有文件和子目录里的所有没有专用侧边栏的文件都会自动添加侧边栏。如果想为子目录添加专用的侧边栏，在子目录中添加一个侧边栏文件即可，这个文件会覆盖顶层侧边栏文件。

4. 不能使用样式表和JavaScript

为了安全，Gollum 会把原始标记文件中的所有 CSS 和 JavaScript 去掉。在命令行中运行 Gollum 时（稍后讨论），可以使用 `--custom-css` 或 `--custom-js` 开关引入自己的 CSS 或 JavaScript 文件，不过托管在 GitHub 中的 Gollum 维基无法引入这些文件。

5. 插入图像

在文档中插入图像的标签，其格式与链接一样，例如 `[[ceo.png]]`。这个标签使用正确的 HTML 标签把名为 ceo.png 的图像插入页面。通常，会在这个标签的基础上增加功能。例如，若想添加边框和 `alt` 属性，可以使用这样的句法：`[[ceo.png|frame|alt=Our CEO relaxing on the beach]]`。这个句法除了为该图像创建正确的 HTML 标签之外，还会添加边框和 alt 文本（有助于更好地理解上下文，并为视障用户使用的读屏软件提供额外的信息）。关于图像众多选项的详细说明，参见 Gollum 仓库的文档。

此外，我们还可以使用 GitHub 的编辑器添加图像。不过你会发现，使用编辑器只能添加图像的地址，不能把图像上传到 GitHub 中（见图 3-3）。

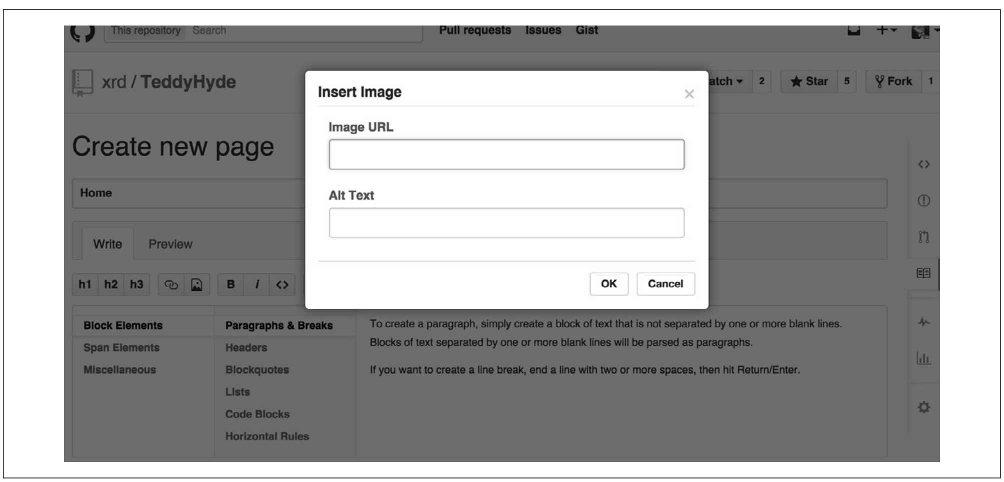


图 3-3：不能上传图像，只能填写图像的 URL

如果非技术用户想为托管在 GitHub 中的 Gollum 维基添加图像，几乎无从下手。接下来，我们通过构建自己定制的以图像功能为核心的 Gollum 编辑器来解决该问题，而且让它仍能兼容常规的 Gollum 维基。非技术用户使用这个编辑器可以上传图像，然后按原有的方

式把维基发布到 GitHub 中。

3.2 改造Gollum

基于 Gollum 开发一个图像编辑器有意义吗？在很多软件团队中，设计团队和软件团队之间存在一个矛盾：设计师通常不喜欢使用源码工具管理图像。当软件开发者所依赖的设计变化迅速时会产生一个问题：程序员很快就跟不上最新的设计了。作为一个维基工具，Gollum 恰好能消除设计师与程序员之间存在的这个矛盾，因为对非技术用户而言，维基易于阅读和修改。由于 Gollum 是个易于改造的维基，我们可以自行构建工作流工具，让设计师便于管理图像，也让程序员易于在源码仓库中查看这些改动。

现在仓库有两个用途。我们可以把 Gollum 仓库当作标准维基，还可以借助我们的应用，以 Gollum 默认界面所没有的更强大的方式输入数据。这样输入的数据仍与 Gollum 兼容，能托管在 GitHub 中。

首先，安装 Gollum 这个 Ruby gem，然后初始化仓库：

```
$ gem install gollum
$ mkdir images
$ cd images
$ git init .
$ printf "### Our home" > Home.md
$ git add Home.md
$ git commit -m "Initial commit"
```

我们刚刚创建了一个与 Gollum 兼容的维基。下面看看这个维基在 Gollum 中是什么样子。执行 `gollum` 命令，然后在浏览器中打开 http://localhost:4567/，如图 3-4 所示。

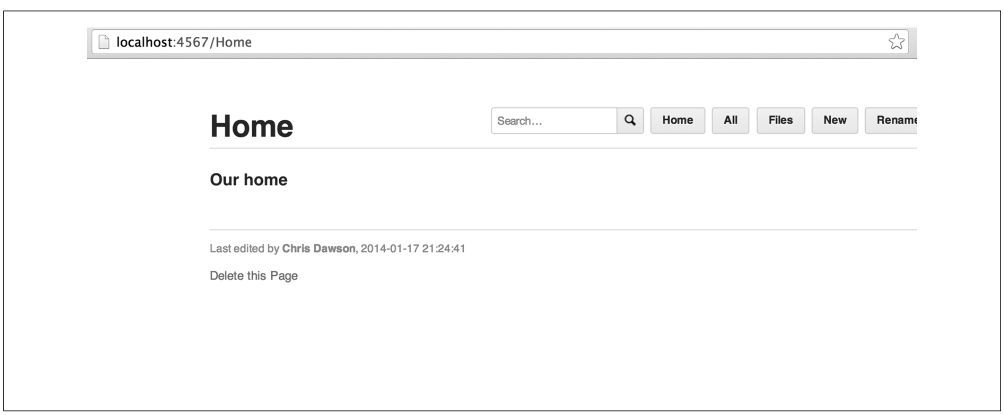


图 3-4：在笔记本电脑中查看维基的首页

可以看出，只需这几个命令就能创建 Gollum 维基的基本结构。

> 如果使用命令行编辑 Gollum 维基，记住，Gollum 只在仓库数据中寻找文件。如果在工作目录中添加了文件，或者没把文件提交到索引中，那么这些文件对 Gollum 是不可见的。

下面开始创建协助我们把图像存储到 Gollum 维基中的 Web 应用。

3.3　开始创建Gollum编辑器

下面开始创建我们定制的编辑器。我们要使用 Sinatra，这是一个 Ruby 库，为构建 Web 应用提供了简单的 DSL（Domain-Specific Language，领域特定语言）。首先，创建一个名为 image.rb 的文件，写入如下内容。

```
require 'sinatra'
require 'gollum-lib'
wiki = Gollum::Wiki.new(".")
get '/pages' do
  "All pages: \n" + wiki.pages.collect { |p| p.path }.join( "\n" )
end
```

然后，创建 Gemfile 文件，安装依赖，最后运行 Web 应用。

```
$ echo "source 'https://rubygems.org'
gem 'sinatra', '1.4.5'
gem 'gollum-lib', '4.1.0'" >> Gemfile
$ bundle install
Fetching gem metadata from https://rubygems.org/..........
Resolving dependencies...
Installing charlock_holmes (0.7.3)
Using diff-lcs (1.2.5)
Installing github-markup (1.3.3)
Using mime-types (1.25.1)
...
$ bundle exec ruby image.rb
$ open http://localhost:4567/pages
```

由于接口和支持库列表都发生了变化，我们指定使用的 `gollum-lib` 至少为 4.1.0 版。最后，执行 `bundle exec ruby image.rb` 命令，在 Bundler 中运行程序（即使用这个 Gemfile 文件安装的 gem，而不是系统中的 gem）。

页面中会列出 Gollum 维基中目前存在的文件。我们只添加了一个文件，即 Home.md。

3.4　以编程的方式处理图像

接下来要把图像添加到服务器中。我们的计划是：把 ZIP 文件上传到系统中，然后解压，把文件添加到仓库中，同时也把这些文件添加到维基中。将 image.rb 脚本改成下面这样。

```
require 'sinatra'
require 'gollum-lib'
require 'tempfile'
require 'zip'
require 'rugged'

def index( message=nil )
  response = File.read(File.join('.', 'index.html'))
  response.gsub!( "<!-- message -->\n",
  "<h2>Received and unpacked #{message}</h2>" ) if message
  response
end

wiki = Gollum::Wiki.new(".")
get '/' do
  index()
end

post '/unpack' do
  @repo = Rugged::Repository.new('.')
  @index = Rugged::Index.new

  zip = params[:zip][:tempfile]
  Zip::Zip.open( zip ) { |zipfile|
    zipfile.each do |f|
      contents = zipfile.read( f.name )
      filename = f.name.split( File::SEPARATOR ).pop
      if contents and filename and filename =~ /(png|jp?g|gif)$/i
        puts "Writing out: #{filename}"
      end
    end
  }
  index( params[:zip][:filename] )
end
```

我们需要 index.html 文件，下面添加：

```
<html>
<body>
<!-- message -->
<form method='POST' enctype='multipart/form-data' action='/unpack'>
Choose a zip file:
<input type='file' name='zip'/>
<input type='submit' name='submit'>
</form>
</body>
</html>
```

这个服务器脚本从挂载点为 `/unpack` 的页面接收 POST 请求，从传入脚本的参数中获取一个 ZIP 文件。然后，这个脚本打开 ZIP 文件（作为临时文件存储在服务器端），迭代里面的各个文件，去掉文件名的完整路径，最后在控制台中把文件名打印出来（如果文件看起来像是图像）。不管是否访问服务器的根目录，也不管是否只向 `/unpack` 挂载点发起 POST

请求，首页始终都要渲染。解压后渲染首页时，要把首页中的一个注释替换成状态消息，以此表明脚本正确接收到了我们发送的文件。

我们要在 Gemfile 文件中添加两个新 Ruby 库（RubyZip 和 Rugged）。使用下述命令添加所需的 gem，然后重新运行那个 Sinatra 服务器脚本。

```
$ echo "gem 'rubyzip', '1.1.7'
gem 'rugged', '0.23.2'" >> Gemfile
$ bundle install
$ bundle exec ruby image.rb
```

Rugged 需要 libgit2 库（使用纯 C 语言实现的访问 Git 仓库的库）。使用 Rugged 修改 Git 仓库能利用 Ruby 语言的优雅，还能获得 C 语言的速度。不过，因为这个库基于 libgit2，而 libgit2 需要 C 编译器，所以安装 Rugged 之前要先安装编译器。在 OS X 中可以执行 `brew install cmake` 命令安装，在 Linux 中可以执行 `apt-get install cmake` 命令安装。

然后，打开 http://localhost:4567/，上传一个全部是图像的 ZIP 文件，测试一下。上传 ZIP 文件之后，在控制台中会看到与下面类似的输出：

```
...
[2014-05-07 10:08:49] INFO  WEBrick 1.3.1
[2014-05-07 10:08:49] INFO  ruby 2.0.0 (2013-05-14)
[x86_64-darwin13.0.0]
== Sinatra/1.4.5 has taken the stage on 4567 for development with
backup from WEBrick
[2014-05-07 10:08:49] INFO  WEBrick::HTTPServer#start: pid=46370
port=4567
Writing out: IMG1234.png
Writing out: IMG5678.png
Writing out: IMG5678.png
...
```

目前，除了打印 ZIP 文件中图像的名称之外，我们什么也没做。下一节，我们将真正地把图像存入 Git 仓库。

3.5　使用Rugged库

这个脚本最终的目标是向 Gollum 维基中添加文件，即把文件添加到 Gollum 维基背后的仓库里。Rugged 库能轻易执行这种繁杂的工作，它是最初操作 Git 的 Ruby 库（Grit）的继任者。写作本书时，Gollum 使用的是 Grit 库，这个库也绑定了 libgit2 库（使用 C 语言实现 Git 核心方法，具有便携性）。Grit 已经被废弃（不过仍有非官方的维护者），因此为了长远发展，Gollum 团队打算把支持 Gollum 的库换成 Rugged。Rugged 使用 Ruby 编写，与 Git 仓库交互时，比原始的 Git 命令更优雅（假如你喜欢 Ruby 的话）。你可能猜到了，Rugged

由 GitHub 的几名员工维护。

为了让那个脚本能修改 Git 仓库，我们不能再让它打印文件名（在解压 ZIP 文件的块中使用 puts 方法），而要调用一个新方法，名为 write_file_to_repo。此外，在解压 ZIP 文件的代码块末尾要调用 build_commit 方法，为新添加的文件构建提交。修改后的脚本如下所示（没有列出文件顶部未改动的代码）：

```
post '/unpack' do
  @repo = Rugged::Repository.new('.')
  @index = Rugged::Index.new

  zip = params[:zip][:tempfile]
  Zip::Zip.open( zip ) { |zipfile|
    zipfile.each do |f|
      contents = zipfile.read( f.name )
      filename = f.name.split( File::SEPARATOR ).pop
      if contents and filename and filename =~ /(png|jp?g|gif)$/i
        write_file_to_repo contents, filename # 写文件
      end
    end
    build_commit() # 为新添加的文件构建提交
  }
  index( params[:zip][:filename] )
end

def get_credentials
  contents = File.read File.join( ENV['HOME'], ".gitconfig" )
  @email = $1 if contents =~ /email = (.+)$/
  @name = $1 if contents =~ /name = (.+)$/
end

def build_commit
  get_credentials()
  options = {}
  options[:tree] = @index.write_tree(@repo)
  options[:author] = { :email => @email, :name => @name, :time => Time.now }
  options[:committer] = { :email => @email, :name => @name, :time => Time.now }
  options[:message] ||= "Adding new images"
  options[:parents] = @repo.empty? ? [] : [ @repo.head.target ].compact
  options[:update_ref] = 'HEAD'

  Rugged::Commit.create(@repo, options)

end

def write_file_to_repo( contents, filename )
  oid = @repo.write( contents, :blob )
  @index.add(:path => filename, :oid => oid, :mode => 0100644)
end
```

从上述代码可以看出，在 Git 仓库中创建提交的大量繁杂工作都由 Rugged 完成。Rugged 提供的接口简单，能在 Git 仓库中创建 blob 文件（write 方法），能把文件添加到暂存

区（add 方法），还能构建树对象（write_tree 方法），然后再构建提交（Rugged::Commit.create）。

为了避免手动输入提交凭据的麻烦，我们实现了 get_credentials 方法，从家目录中的 .gitconfig 文件里读取凭据。只要设备中安装了 Git，这个文件可能就存在；如果没有这个文件，get_credentials 方法会失效。在我的设备中，.gitconfig 文件的内容如下述代码片段所示。get_credentials 方法的作用很简单：加载并解析 .gitconfig 文件，获取用户名和电子邮件地址。如果想使用其他方式获取凭据，或者想手动提供，可以修改 get_credentials 方法，满足自己的需求。build_commit() 方法会使用实例变量 @email 和 @name。

```
[user]
        name = Chris Dawson
        email = xrdawson@gmail.com
[credential]
        helper = cache --timeout=3600
...
```

下面验证上传 ZIP 文件后是否能正确处理。上传文件后，打开终端窗口，执行下述命令：

```
$ git status
```

让人奇怪的是，我们会看到如下输出情况：

```
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

  deleted:    images/3190a7759f7f668.../IMG_20120825_164703.jpg
  deleted:    images/3190a7759f7f668.../IMG_20130704_151522.jpg
  deleted:    images/3190a7759f7f668.../IMG_20130704_174217.jpg
```

我们是添加文件啊，为什么 Git 报告说文件被删除了呢？

为了弄明白怎么回事，要知道在 Git 中文件有三个存储区：工作目录、暂存区和仓库。工作目录中存储的是正在处理的本地文件。根据文档，git status 命令的作用是“显示工作区的状态”。Rugged 操作的是仓库本身，上述代码中的 Rugged 方法调用操作的是暂存区，然后构建提交。这一点要特别注意，因为如果只使用 Rugged 方法调用执行写操作，文件不会存入工作目录，此时，在本地运行的 Gollum 中，维基页面里无法引用那些文件。下一节会修正这个问题。

现在，我们把文件添加到仓库中了，可是还没有把文件提供给维基。下面，我们要修改这个服务器脚本，把各个文件写入一个维基页面中，以供查看。如前所述，文件既要写入工作区，也要写入仓库（使用 Rugged 库中的 write 方法）。然后，我们可以生成一个 Review

文件，列出上传的所有图像。

3.6 优化图像存储

如果设计师上传相同的图像两次会怎样？我们的代码根据父级仓库的 SHA 散列值把上传的图像写入硬盘中的某个路径（因此，即使与之前上传的文件相同，也会写入不同的路径）。对外行人来说，这看似是多次添加相同的文件。然而，根据 Git 的特性，相同的文件可以多次添加，而且首次添加之后，再添加相同的文件不会消耗额外的存储空间（只会稍微增加树结构的大小）。向 Git 仓库添加文件时，Git 会根据文件的内容生成 SHA 散列值。例如，空文件对应的 SHA 散列值始终相同：[2]

```
$ echo -en "blob 0\0" | shasum
e69de29bb2d1d6434b8b29ae775ad8c2e48c5391
$ printf '' | git hash-object -w --stdin
e69de29bb2d1d6434b8b29ae775ad8c2e48c5391
```

如果后面上传的 ZIP 文件中只有一两个文件与之前上传的 ZIP 文件不同，Git 能正确地多次引用相同的文件。可惜，GitHub 没像常规的仓库那样为维基提供查看统计资料的界面。不过，在本地仓库中可以执行 Git 子命令 `count-objects` 查看仓库的大小。例如，上传包含两个图像的 ZIP 文件之后，我执行了 `count-objects` 命令，看到如下输出：

```
$ git gc
...
$ git count-objects -v
count: 0
size: 0
in-pack: 11
packs: 1
size-pack: 2029
prune-packable: 0
garbage: 0
size-garbage: 0
```

审查第一个 ZIP 文件，看到如下统计资料：

```
$ unzip -l ~/Downloads/Photos\ \(4\).zip
Archive:  /Users/xrdawson/Downloads/Photos (4).zip
  Length     Date   Time    Name
 --------    ----   ----    ----
  1189130  01-01-12 00:00   IMG_20130704_151522.jpg
   889061  01-01-12 00:00   IMG_20130704_174217.jpg
 --------                   -------
  2078191                   2 files
```

下面我们再审查一个 ZIP 文件，其中有两个文件与之前相同，此外又添加了一个图像文件：

注 2：这篇博文对此做了极好的说明：http://alblue.bandlem.com/2011/08/git-tip-of-week-objects.html。

```
unzip -l ~/Downloads/Photos\ \(5\).zip
Archive:  /Users/xrdawson/Downloads/Photos (5).zip
  Length     Date   Time    Name
 --------    ----   ----    ----
  1189130  01-01-12 00:00   IMG_20130704_151522.jpg
   566713  01-01-12 00:00   IMG_20120825_164703.jpg
   889061  01-01-12 00:00   IMG_20130704_174217.jpg
 --------                   -------
  2644904                   3 files
```

然后，上传第二个 ZIP 文件。再次执行 `count-objects` 命令（在 `git gc` 命令之后执行，这个命令用于高效打包文件，让输出更易于阅读），看到如下输出：

```
$ git gc
...
$ git count-objects -v
count: 0
size: 0
in-pack: 17
packs: 1
size-pack: 2578
prune-packable: 0
garbage: 0
size-garbage: 0
```

注意，包大小只增加了约 0.5MB，即那第三个文件压缩后的大小。而更应该注意的是，虽然另外两个文件从不同的路径添加，但是对仓库的大小没有影响。

如果再次上传第二个 ZIP 文件，我们要重新生成并提交 Review.md 文件的一个新版本，但是无需读取图像目录，然后在 Git 仓库中新建文件（尽管文件的路径变了），因此对仓库的影响微乎其微：

```
$ git gc
...
$ git count-objects -v
count: 0
size: 0
in-pack: 21
packs: 1
size-pack: 2578
prune-packable: 0
garbage: 0
size-garbage: 0
```

可以看出，包大小几乎没变，这表明只新增了一个 Git 树对象和一个提交对象。在仓库中，这些文件放在不同的路径，因此不管访问的是哪个版本，查看页面都能正确将其显示出来。

```
$ find images
images
```

```
images/7507409915d00ad33d03c78af0a4004797eec4b4
images/7507409915d00ad33d03c78af0a4004797eec4b4/IMG_20120825_164703.jpg
images/7507409915d00ad33d03c78af0a4004797eec4b4/IMG_20130704_151522.jpg
images/7507409915d00ad33d03c78af0a4004797eec4b4/IMG_20130704_174217.jpg
images/7f9505a4bafe8c8f654e22ea3fd4dab8b4075f75
images/7f9505a4bafe8c8f654e22ea3fd4dab8b4075f75/IMG_20120825_164703.jpg
images/7f9505a4bafe8c8f654e22ea3fd4dab8b4075f75/IMG_20130704_151522.jpg
images/7f9505a4bafe8c8f654e22ea3fd4dab8b4075f75/IMG_20130704_174217.jpg
images/b4be28e5b24bfa46c4942d756a3a07efd24bc234
images/b4be28e5b24bfa46c4942d756a3a07efd24bc234/IMG_20130704_151522.jpg
images/b4be28e5b24bfa46c4942d756a3a07efd24bc234/IMG_20130704_174217.jpg
```

Git 和 Gollum 无需重载仓库就能把相同的文件存储在不同的路径上。

3.7 在GitHub中查看

维基的作用是注解开发项目。如果你按照前文所说的做了，而且为仓库新建了维基，那么你可以使用 `image.rb` 脚本推送所做的改动。新建维基之后，在页面右边寻找“Clone this wiki locally”文本框，如图 3-5 所示。

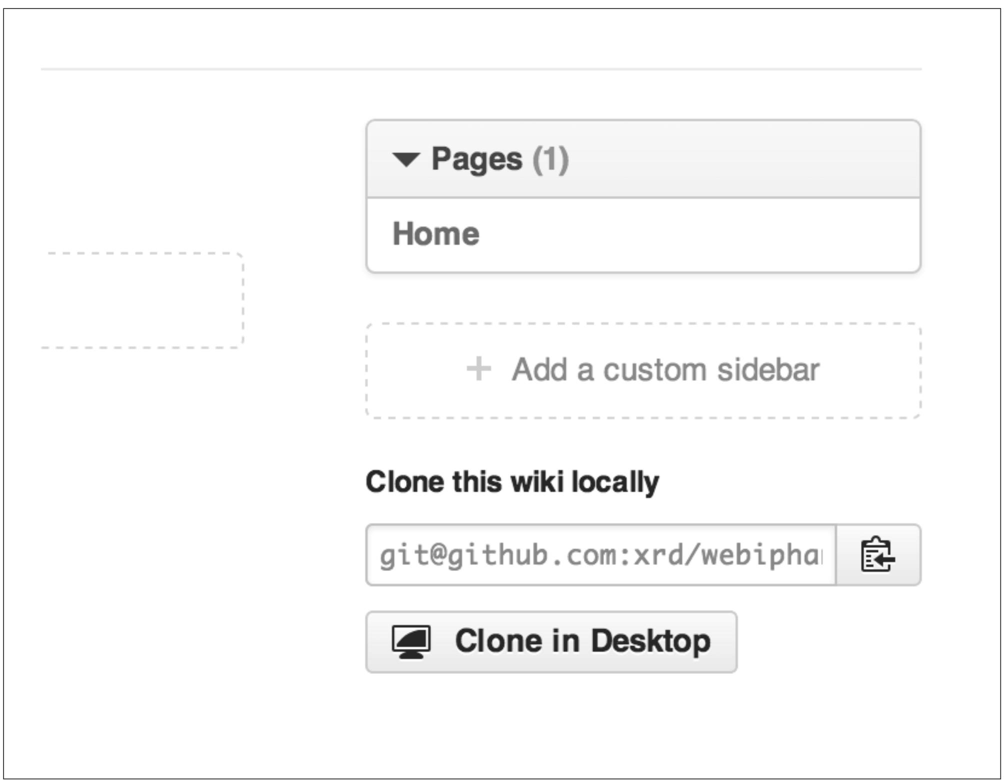


图 3-5：找到维基的克隆 URL

复制那个链接，然后打开终端窗口。我们将在终端里为本地仓库添加远程 URL，这样便可以同步仓库，把图像发布到 GitHub 中。Gollum 维基的 URL 结构简单，只是在仓库的克隆 URL 后面加上 `.wiki`（不过是在末尾的 `.git` 扩展名之前）。因此，如果仓库的克隆 URL 是 `git@github.com:xrd/webiphany.com.git`，那么相应的维基的克隆 URL 便是 `git@github.com:xrd/webiphany.com.wiki.git`。知道 URL 之后，可以使用下述命令为本地仓库添加远程仓库地址：

```
$ git remote add origin git@github.com:xrd/webiphany.com.wiki.git
$ git pull # 这个命令会要求你合并改动……
$ git push
```

拉取仓库后，Git 会要求你合并改动，因为本地仓库中没有 GitHub 创建的 Home.md 文件。此时，直接合并即可。后面的 `git push` 命令用于发布改动。现在访问维基，会发现右边的 Pages 侧边栏中多了一个文件。点击 Review，在页面中可以看到我们最近添加的图像，如图 3-6 所示。

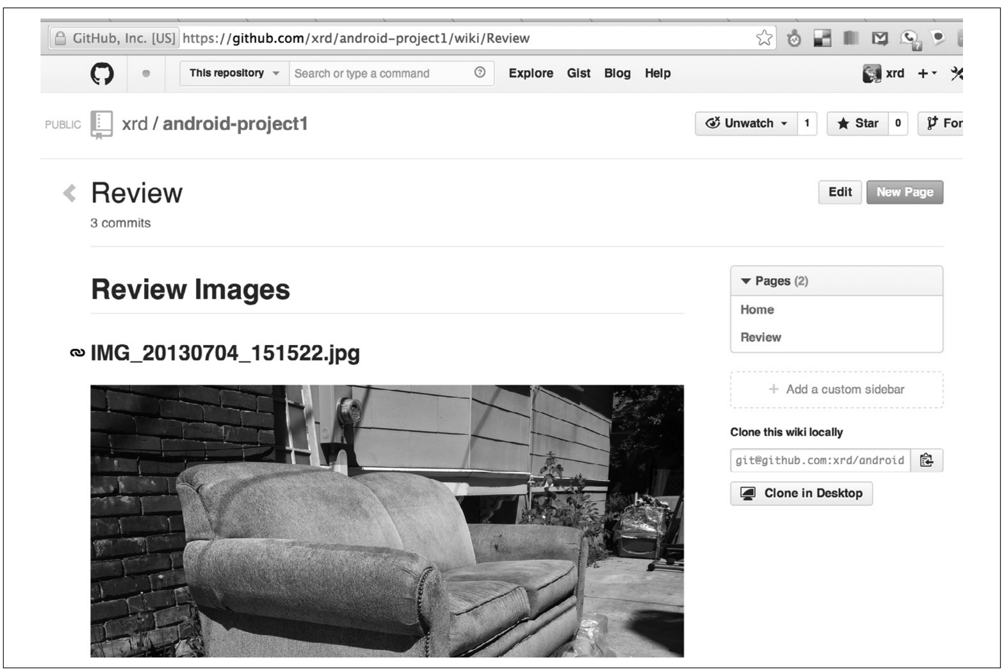


图 3-6：查看图像的页面

我不确定设计师提供一张沙发图的原因，不过他肯定有他自己的理由。

发布文件之后，我们可以点击侧边栏中的 Review 链接，查看 Review 页面的最新版。此外，还可以点击页面标题正下方的“3 Commits”（其中的数字会根据文件的提交次数变动），查看这个文件的修订版本。在打开的页面中有文件的完整历史，如图 3-7 所示。

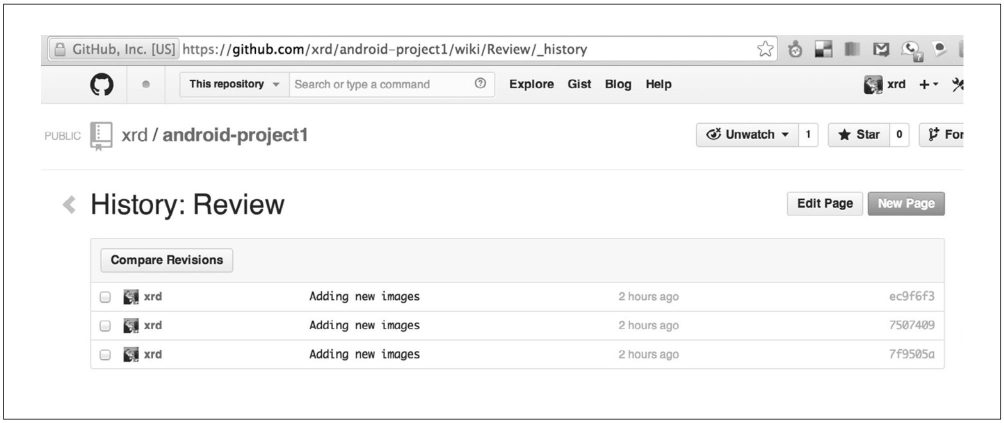


图 3-7：通过提交日志查看维基的历史

点击 SHA 散列值可以查看页面相应的修订版本，以及文档在彼时的状态。可惜，在修订版本之间切换需要点击两次，一次从 Review 页面到修订版本列表，另一次是进入想查看的修订版本。不过这么做可以查看设计师提供的图像素材变化情况。

如果 GitHub 提供一种便捷的方式，可以从一个修订版本转到父级（之前的）修订版本就好了，可惜现在还未提供。不过我们可以自己动手，在 Review 页面中添加一个特殊的链接，通过特定的方式，让它指向页面的前一版。

3.8 改善修订版本导航

在我们的示例中，现在只有三个修订版本，而且都使用相同的提交说明（“Adding new images”）。这种说明不够清晰，也不便于区分不同的修订版本，但是其对于理解图像素材的变化情况是很重要的。我们可以轻松地改善这种不便。

首先，在上传表单中添加一个字段，用于输入提交说明：

```
<html>
<body>
<!-- message -->
<form method='POST' enctype='multipart/form-data' action='/unpack'>
Choose a zip file:
<input type='file' name='zip'/>
<input type='text' name='message' placeholder='Enter commit message'/>
<input type='submit' name='submit'>
</form>
</body>
</html>
```

然后，通过修改一行选项散列，即把提交说明设为通过参数传入的值，来修改 image.rb 脚本里的提交说明：

```
  ...
options[:committer] = { :email => @email, :name => @name, :time => Time.now }
options[:message] = params[:message]
options[:parents] = @repo.empty? ? [] : [ @repo.head.target ].compact
  ...
```

现在，设计师发布新的 UI 素材时可以指明改动了什么，而改动会记录在案，能通过 GitHub 的修订历史页面查看。

3.9 修缮素材页面之间的链接

如前所述，进入 Review 页面的某个修订版本之后，无法快速在版本之间切换。不过，你应该还记得我们前面使用父级 SHA 散列值构建过图像的链接。同理，在某个修订版本页面查看历史时，可以在图像素材页面插入一个导航链接。

这一次要改的也不多，只需修改 `write_review_file` 方法中的一行代码。在创建指向各个图像文件链接的那个代码块之后增加一行，通过 `@repo.head.target` 获得 Rugged 对象中的父级 SHA 散列值，然后构建指向父级文档的链接。这个链接用于访问历史中的前一个修订版本：

```
  ...
files.each do |f|
  contents += "### #{f} \n[[#{dir}/#{f}]]\n\n"
end
contents += "[Prior revision (only when viewing history)]" +
"(#{@repo.head.target})\n\n"

File.write review_filename, contents
oid = @repo.write( contents, :blob )
  ...
```

现在，查看 Review 文件的历史时，会看到指向之前版本的链接。那么，能不能添加指向历史中下一个版本的链接呢？可惜，我们不能预料到仓库中下一个提交的 SHA 散列值，因此这个 Ruby 脚本无法在 Review.md 文件中构建这样的链接。不过，我们可以变通一下，使用浏览器的后退按钮回到浏览历史中的前一个页面。你可能想更进一步，在链接中使用 JavaScript，调用 `window.history.back()`，但是 Gollum 会从渲染后的标记文件中去掉 JavaScript，因此这样行不通。一般来说，这是正确的处理方式，因为我们不希望维基页面中出现恶意标记；可是对这个问题而言，却限制了选择。

可惜，直接查看 Review 文件时，这种链接无效（如果点击，会把你带到一个页面，让你新建页面）。Gollum 与 Jekyll 不同，它不支持 Liquid 标签；如果支持的话，可以使用用户名和仓库名构建链接。目前，我们无法访问这些变量，因此要使用相对链接，所以链接只在查看 Review 页面的历史时有效，而直接访问 Review 页面时无效。这个局限不影响查看文件，不过还是要让相关人员知道。

3.10 小结

本章，我们学习了如何从头开始创建 Gollum 维基，包括在 GitHub 中创建，以及使用命令行创建新仓库的方式。然后，我们介绍了 Gollum 命令行工具的多种用法，了解到自己运行 Gollum 服务器时使用命令行工具的好处。最后，我们围绕图像功能，使用 Rugged 和 Sinatra 两个 Ruby 库定制了一个 Gollum 图像编辑器。

下一章换个完全不同的话题：构建一个搜索 GitHub 工单的 GUI 应用。我们将使用 Python 构建该应用。

第4章

Python和Search API

数据多到一定程度后，再大的组织也不可能做到让一切轻松可查。Google 告诉我们，当数据达到一定规模后，唯一有效的系统是：搜索框。在 GitHub 中有两种仓库：一种是你能直接访问的（数量较少），这是一种单层结构，在脑中就能直接记住；另一种是数百万个数也数不完的其他人创建的公开仓库，你可以通过搜索框提供的强大功能找到所需的内容。

令人欣喜的是，GitHub 还通过 API 开放了搜索功能，以便在你自己的应用中使用。通过 GitHub 的 Search API，我们能使用所有的内建搜索功能，包括逻辑和作用域运算符，如 `"or"` 和 `"user"`。你可以在自己的应用中集成这个功能，为用户提供十分强大的搜索功能，让他们找到自己所需的内容。

本章将深入了解这个 API，并尝试使用它构建一个有用的应用。我们将说明 Search API 的结构，返回的结果有哪些类型，以及如何利用它改善我们团队中某个人的工作。

4.1　Search API概述

Search API 由四部分组成：仓库、代码、工单和用户。这些 API 的作用不同，结果的格式也不同，不过主要行为基本一致。我们先讨论相同点，这样便于理解后文对各个 API 返回结果的说明。这些 API 主要有四个共同特征。

4.1.1　身份验证

用户的身份对搜索词条得到的结果有影响，因此一定要知道身份验证方面的知识。我们在

1.6 节详述了 GitHub 验证身份的方式。不登录也能使用 Search API，不过有几个限制。

首先，只能搜索公开仓库。如果只与开源软件打交道，这或许不算是限制；可是，你的应用的用户可能想访问他们自己的以及任何所属组织的私有代码。此外，所有企业版仓库都是私有的，匿名搜索没有任何作用。

其次，验证身份后频率限制得到提升。搜索的频率限制比其他 API 严格，因为搜索要消耗很多计算资源，因此对匿名搜索的限制更严。写作本书时，根据文档，对匿名搜索的限制是每分钟 5 次，而验证身份后，每分钟可查询 20 次。对频率限制的详细说明参见 1.7.6 节。

下面以表格的形式列出：

	匿　　名	验　证　后
结果中包含私有仓库	否	是
可以在企业版中使用	否	是
频率限制	每分钟 5 次	每分钟 20 次

4.1.2　结果的格式

不管搜索什么，Search API 返回的结果都是特定的格式。下面是某次查询的结果示例，我对其做了大量节略，以便让你看清始终存在的部分。

```
{
  "total_count": 824,
  "incomplete_results": false,
  "items": [
    {
      …
      "score": 3.357718
    }
  ]
}
```

从上往下看，`total_count` 字段表示此次查询得到的搜索结果总数。搜索结果通常有几千个，毕竟 GitHub 中有几百万个仓库。默认只返回前 30 个搜索结果，不过可以在 URL 中使用 `page` 和 `per_page` 两个查询参数定制。例如，向下述 URL 发起 GET 请求会得到 45 个结果，而且从第 46 个算起：

```
search/repositories?q=foobar&page=2&per_page=45
```

通常，每页限制返回 100 个结果。

`incomplete_results` 字段表示 GitHub 对 Search API 所做的计算限制。如果搜索耗时太长，那么 GitHub API 会在执行中途停止，返回已经得到的结果，并把这个标志设为 `true`。大

多数情况下，这不会导致问题，`total_count` 字段的值是搜索结果的总数；然而，如果是复杂的词条，可能只会得到部分结果。

搜索结果是一个数组，存在 `items` 字段中，而且各个元素都有 `score` 字段。这个字段的值是数字，不过只是结果与词条匹配程度的相对值，用于默认排序，分值高的在前面。如果对这个字段感兴趣，要记住，仅当与同一个词条的其他结果比较时，它才有意义。得到 50 分的结果不一定比得到 5 分的结果“好”十倍。

下表总结了重要字段的意义。

字　　段	意　　义
`total_count`	搜索结果总数
`incomplete_results`	如果搜索在结束前中止，值为 `true`
`items`	搜索结果列表
`(item).score`	单个搜索结果的相关度

4.1.3　搜索运算符和限定符

当然，最好能避免分页，或者至少在第一页返回最好的结果。限定符和运算符有助于删减搜索结果，减少分页数量，有时还能把正确的结果移到顶部。

对 Search API 来说，所有搜索都要有搜索词条，而这个词条要经过编码，通过 URL 中的 `q` 参数传入。大多数词条是自由词，不过这个 API 还支持一些更强大的句法，例如下述形式。

- `x AND y`，还可以使用 `OR` 或 `NOT`
- `user:<name>`，其中 *name* 是用户或组织的名称
- `repo:<name>`
- `language:<name>`
- `created:<date(s)>`
- `extension:<pattern>`，匹配文件扩展名（如 py 或 ini）

数值和日期可以指定范围：

- `2015-02-01` 只匹配指定的日期
- `<2015-02-01` 匹配指定日期之前的任何日期
- `2015-02-01..2015-03-01` 匹配指定范围内的日期，包括两端点

例如，如果想查找用户 tpope 在 2012 年 7 月编写的代码，查询参数可以写为 `"user:tpope created:2012-07-01..2015-07-31"`。编码后的 URL 如下：

```
search/repositories?q=user%3Atpope+created%3A2012-07-01..2015-07-31
```

如果只想搜索 Python 代码，那么可以添加代码 `language=python`，在 URL 后面加上编码 `+language%3Apython`。

此外还有其他选项。https://github.com/search/advanced 页面提供了一个 UI，能帮助你构建一条查询。

4.1.4 排序

使用搜索词条运算符删减后，得到的结果如果还不是最重要的元素，或许可以再做排序。搜索结果使用特定的顺序返回，而不是随机的。默认按“最佳匹配”排序，即按照搜索得分排序结果，得分高的在前面。如果想改变排序依据，可以把 `sort` 查询参数设为 `stars`、`forks` 或 `updated`，例如 `search/repositories?q=foobar&sort=stars`。

此外，还可以使用 `order` 参数逆向排序，例如 `search/repositories?q=foobar&sort=stars&order=desc`。默认的排序方式是 `desc`（降序），不过也可以设为 `asc` 来逆向排序。

4.2 Search API详解

至此，我们已经概述了这些 API 的相同点，下面详述细节。Search API 分为四类：仓库、代码、工单和用户。这四类的基本机制是一样的：向端点发起 GET 请求，通过 `q` 参数传入经过 URL 编码后的搜索词条。下面分述各个 API 删减后的响应，同时说明部分值得关注的字段。

4.2.1 搜索仓库

`search/repositories` 端点通过仓库的元数据查找匹配词条的仓库。默认使用的元数据是项目的名称和描述，不过还可以在词条中指定 `in:readme`，搜索自述文件。其他限定符参阅文档：https://developer.github.com/v3/search/#search-repositories。

`search/repositories?q=foobar` 查询可能会得到如下响应：

```
{
  "total_count": 824,
  "incomplete_results": false,
  "items": [
    {
      "id": 10869370,
      "name": "foobar",
      "full_name": "iwhitcomb/foobar",
      "owner": {
        "login": "iwhitcomb",
        "id": 887528,
        "avatar_url": "https://avatars.githubusercontent.com/u/887528?v=3",
        …
```

```
      },
      "private": false,
      "html_url": "https://github.com/iwhitcomb/foobar",
      "description": "Drupal 8 Module Example",
      "fork": false,
      …
      "score": 59.32314
    },
    …
  ]
}
```

`items` 中的各个元素是仓库的描述信息，包罗万象，例如仓库界面的 URL（`html_url`）、属主的头像（`owner.avatar_url`），以及适合使用 Git 克隆仓库的 URL（`git_url`）。

4.2.2 搜索代码

`search/code` 端点用于搜索仓库里的内容。我们可以在文件的内容中查找匹配内容，也可以在文件的路径中查找（使用 `in:path`）。（其他限定符参阅文档：https://developer.github.com/v3/search/#search-code。）

这个 API 有几个限制是其他搜索端点没有的，因为服务器要排查大量数据寻找匹配内容。第一，要提供一般性的搜索词条（要匹配的短语）；其他几个 API 在词条中可以只指定运算符（如 `language:python`），但是这个 API 不可以。第二，词条中所有的通配符都会被忽略。第三，不会搜索超过一定大小的文件。第四，只搜索项目的默认分支，通常为 `master`。第五，这可能是最重要的一点，*必须*使用 `user:<name>` 限定符指定仓库的属主，不能一次搜索全部仓库。

返回的 JSON 如下所示：

```
{
  "total_count": 9246,
  "incomplete_results": false,
  "items": [
    {
      "name": "migrated_0000.js",
      "path": "test/fixtures/ES6/class/migrated_0000.js",
      "sha": "37bdd2221a71b58576da9d3c2dc0ef0998263652",
      "url": "…",
      "git_url": "…",
      "html_url": "…",
      "repository": {
        "id": 2833537,
        "name": "esprima",
        "full_name": "jquery/esprima",
        "owner": {
          "login": "jquery",
          "id": 70142,
```

```
          "avatar_url": "https://avatars.githubusercontent.com/u/70142?v=3",
          …
        },
        "private": false,
        …
      },
      "score": 2.3529532
    },
    …
  ]
}
```

各个元素中有关于匹配文件的数据，例如文件名和文件的几种 URL。后面还有文件所属仓库的数据。最后是得分，用于默认的“最佳匹配”排序。

4.2.3 搜索工单

仓库中不仅有代码。`search/issues` 端点在项目中搜索匹配词条的的工单和拉取请求。这个端点可以使用的搜索限定符众多，举例如下。

- `type`

  值为表示拉取请求的“pr”，或者表示工单的“issue”（默认兼具二者）。

- `team`

  匹配讨论中提到指定团队的工单（仅用于在你所属的组织中搜索）。

- `no`

  匹配没有某个数据的工单（如“no:label”）。

此外还有很多，详情参阅文档：https://developer.github.com/v3/search/#search-issues。

请求这个端点得到的响应如下：

```
{
  "total_count": 1278397,
  "incomplete_results": false,
  "items": [
    {
      "url": "…",
      "labels_url": "…",
      "comments_url": "…",
      "events_url": "…",
      "html_url": "…",
      "id": 69671218,
      "number": 1,
      "title": "Classes",
      "user": {
        "login": "reubeningber",
        "id": 2552792,
```

```
        "avatar_url": "…",
        …
      },
      "labels": [
        …
      ],
      "state": "open",
      "locked": false,
      "assignee": null,
      "milestone": null,
      "comments": 0,
      "created_at": "2015-04-20T20:18:56Z",
      "updated_at": "2015-04-20T20:18:56Z",
      "closed_at": null,
      "body": "There should be an option to add classes to the ul and li...",
      "score": 22.575937
    },
  ]
}
```

同样，列表中的各个元素是查询这个 API 得到的结果。这里有很多有用的数据，例如工单的标题（`title`）、标注（`labels`），以及指向拉取请求相关数据的链接（`pull_request.url`，如果结果不是拉取请求，没有这个字段）。

4.2.4　搜索用户

前面几个 Search API 都与仓库有关，但是这个端点搜索的是一个不同的命名空间——GitHub 用户。默认情况下只搜索用户的登录名和公开的电子邮件地址。`in` 限定符可以扩大搜索范围，包含用户的全名：`in:fullname,login,email`。此外还有几个有用的限定符，详情参阅文档：https://developer.github.com/v3/search/#search-users。

查询 `search/users` 端点得到的响应如下：

```
{
  "total_count": 26873,
  "incomplete_results": false,
  "items": [
    {
      "login": "ben",
      "id": 39902,
      "avatar_url": "…",
      "gravatar_id": "",
      "url": "…",
      "html_url": "…",
      …
      "score": 98.24275
    },
    {
      "login": "bengottlieb",
      "id": 53162,
```

```
        "avatar_url": "…",
        "gravatar_id": "",
        "url": "…",
        "html_url": "…",
        …
        "score": 35.834213
      },
    ]
  }
```

这里的元素列表是查询 `users/<name>` 端点得到的结果。有用的字段有用户的头像（`avatar_url`），指向其他 API 端点的几个链接（`repos_url`，`url`），以及结果的类型（用户或组织，在 `type` 字段中标明）。

4.3 示例应用

现在，我们大致了解了 Search API 的行为，下面用它做些有用的事。

假设你的开发团队把 Git 仓库存储在 GitHub 中，而且把运行应用所需的组件分成多个小仓库。这种情况对于非技术类同事来说相当棘手，如果想报告问题，那么他们不知道该发到哪个仓库里，而且也不知道怎么查找报告过的问题。

此时可以使用搜索，但是在组织的所有仓库中搜索要使用 `user:<organization>` 运算符，而非程序员遇到这种情况时有点害怕，不能立即写出运算符。此外，用户每次搜索工单时都要记得添加这个选项。

Search API 能稍微简化这种操作。我们将构建一个 GUI 应用，界面中只有一个搜索框，这样非技术类用户能特别轻易地搜索某个组织中所有仓库的所有工单。最终构建得到的应用如图 4-1、图 4-2 和图 4-3 所示。

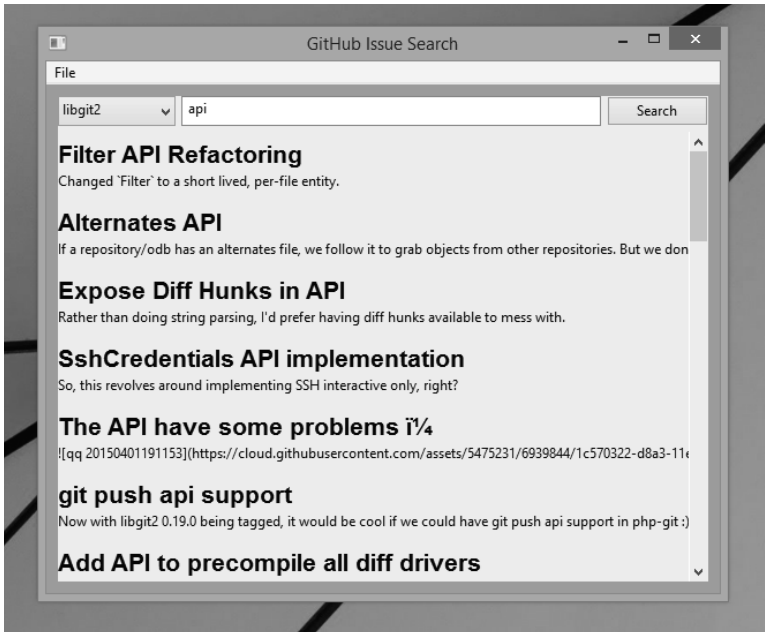


图 4-1：在 Windows 中运行搜索 GitHub 的应用

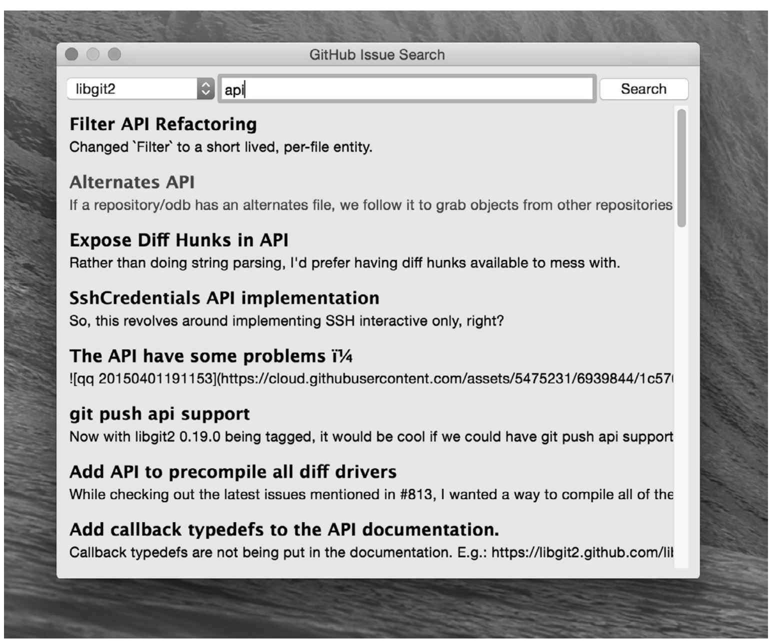


图 4-2：在 Mac 中运行搜索 GitHub 的应用

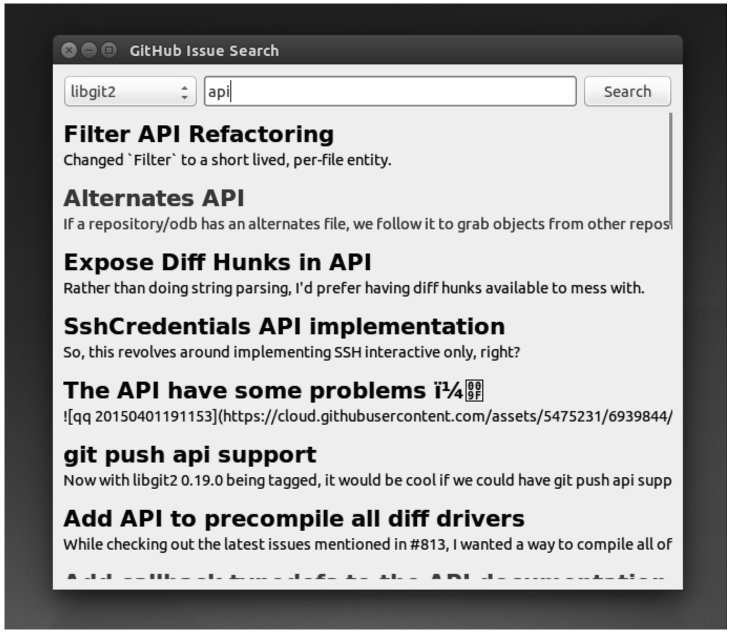


图 4-3：在 Linux 中运行搜索 GitHub 的应用

用户流

以上是最终目标，下面分析这个应用的用户体验。

首先，我们要让用户使用 GitHub 凭据登录。为什么？部分原因是 Search API 有特别严格的限流措施，而且验证身份的用户具有更高的频率限制。此外，还因为要让用户搜索私有仓库的工单。为了简化验证过程，我们的程序首先尝试从 Git 的凭据库中获取 GitHub 凭据，如果不成功再显示登录表单，如图 4-4 所示。

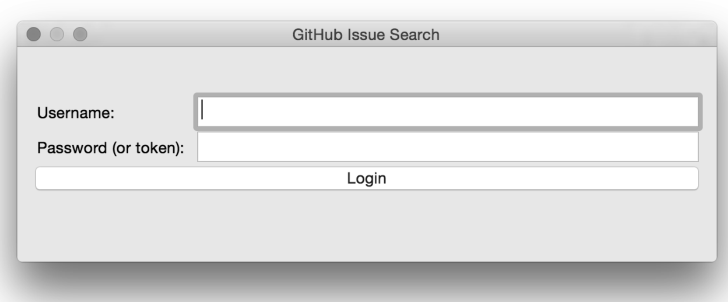


图 4-4：登录界面

用户登录后，会显示一个搜索框。输入搜索词条，然后按回车键，界面中会显示一个滚动列表，列出搜索结果，显示工单的标题和描述的第一行内容。点击某个搜索结果后，在用户的浏览器中打开对应的工单。

就这么简单。对用户来说，这个应用只有两个主界面。这是一个简单而专注的工具，用于解决十分明确的问题，因此代码应该不太难。

4.4 Python

现在我们知道这个程序的行为了，接下来要决定怎么实现。

基于以下几个原因，我们将使用 Python 语言实现。首先，目前为止本书还未提到这门语言，而我想让你接触各种语言。本书的目标之一是带着读者探索之前未曾见过的技术。

其次，有个用于构建 GUI 应用的 Python 库，无需修改就能在 Mac OS X、Linux 和 Windows 中运行。对一门现代的高级编程语言来说，具有如此独特的功能真是让人惊喜。如果想通过别的技术实现兼容性，往往要使用十分复杂的框架、底层语言（如 C++），或者二者兼用。

最后，使用 Python 构建的应用易于分发。有个 Python 包能把整个 Python 程序及其所有的依赖打包成一个文件（在 OS X 中打包成 .app 文件）。因此，若想把这个程序提供给同事，通过电子邮件发送一个 ZIP 文件即可。这对我们的案例而言是有帮助的，因为非技术类用户可能并不乐于使用安装程序（有时甚至无权在设备中安装程序）。

下面概览编写这个应用的代码要使用的库。稍后会说明具体怎么用，在此之前先概览一下有助于理解各个库的作用。使用 Python 做开发往往会遇到这样的窘境：不同包的安装方式不同，所以我还会说明如何安装各个库。

4.4.1 AGitHub

首先介绍与 GitHub API 交互的库，即 `agithub`。这个库抽象的层次不深，它把 GitHub 的 REST API 转换成在对象上调用的方法，写出的代码优雅且易于阅读。

`agithub` 库的项目地址是 https://github.com/jpaugh/agithub，安装方法很简单，只需下载一份 agithub.py 源码文件的副本，与项目的文件放在一起即可。

4.4.2 WxPython

WxPython 用于创建应用的图形界面。这个库使用 Python 对低层的 WxWidgets 工具包做了面向对象抽象。WxWidgets 是原生 UI 工具包的通用代码适配器，支持 Linux、Mac 和 Windows 操作系统的原生控件，因此可以使用相同的 Python 代码与这些平台交互。

WxPython 项目的网站是 http://www.wxpython.org，那里有详细信息。页面的左手边有针对各个平台的下载链接。下一版 WxPython（代号“Phoenix”）将能通过 pip 安装，不过写作本书时 Phoenix 只有预先发布版，尚不稳定，最好别用。

> 稍微说一下 Python 的背景。现在，Python 处于过渡期，有两个广泛使用的版本，即 Python 2.7 和 Python 3（写作本书时是 3.5 版）。这里你无须关心两个版本之间的大多数差异细节，如果想跟着这个示例一起做，那就使用 Python 2.7，因为 WxPython 目前尚不支持 Python 3。即将发布的 Phoenix 计划支持 Python 3。不过编写后文的代码时考虑了向前兼容，因此如果你阅读本书时 Phoenix 已经发布，在 Python 3 中运行也不会遇到什么问题。

4.4.3 PyInstaller

PyInstaller 是我们要使用的分发工具。这个程序的主要功能是读取 Python 代码，然后分析，找出全部依赖，最后把所有文件（包括 Python 解释器）放入一个目录里。这个程序甚至还能打包那个目录，这样双击包就能运行程序。所有这些操作无需过多人工干预，配置选项也没几个。写过 GUI 应用的人都知道这些问题很难处理。

如果想了解这个项目的信息，请访问 http://pythonhosted.org/PyInstaller。这个程序可以使用 Python 的包管理器安装，方法是执行 `pip install pyinstaller` 命令。

4.5 编写代码

好了，现在我们知道要使用 Python 生态系统中的哪些库了。下面开始编写代码，利用那些库构建一个应用。首先，编写下述骨架文件。

```
#!/usr/bin/env python ❶

import os, subprocess
import wx
from agithub import Github ❷

class SearchFrame(wx.Frame): ❸
    pass

if __name__ == '__main__': ❹
    app = wx.App() ❺
    SearchFrame(None)
    app.MainLoop()
```

注意以下几点。

❶ `#!` 字符序列表明这是一个 Python 2.7 程序。

❷ 导入要使用的库。我们把 WxPython（wx）全部导入，但是对 agithub 来说，只需要 Github 类（注意大小写）。os 和 subprocess 出自 Python 标准库。

❸ 这是实现主窗口的类。讨论具体的实现时再细说。

❹ 在 Python 中要使用这种句法创建应用的主入口点。

❺ 在 WxPython 中，主函数要这么写。我们实例化一个 App 实例，创建顶层窗体的实例，然后运行应用的主循环。

下载运行这个程序，你会看到命令行挂起了，其实它是在等待 GUI 输入。这是因为 wx 库无法创建没有内容的窗体。下面我们来修正这个问题，不过在此之前先岔个题，了解一些 Git 的内部机制，以便提升用户体验。

4.5.1　获取Git凭据的辅助函数

前面的代码清单是大多数 UI 代码的基本结构，在实现具体细节之前应该定义一个函数，获取用户的 GitHub 凭据。我们会取个巧，先询问 Git 有没有保存用户的登录名和密码。

为此，要使用 git credential fill 命令。Git 内部就是这么做的，这样与 GitHub 中的远程仓库交互时无需用户每次都输入 GitHub 网站的密码。这个命令的运作方式是，从 stdin 中以 <key>=<value> 格式接收关于连接的所有信息，调用方提供所需的全部信息后，关闭 stdin 流（或者输出一个空行），然后 Git 输出所获的关于连接的全部信息。运气好的话，这些信息中会有用户的登录名和密码。整个交互过程有点像下面这样：

```
$ echo "host=github.com" | git credential fill ❶
host=github.com
username=ben ❷
password=(redacted)
```

❶ 把一行输入传给 git credential，然后关闭 stdin，告诉 Git 输入结束。

❷ Git 输出所获的关于连接的全部信息，包括输入的值，以及用户名和密码（如果 Git 知道的话）。

关于 git-credential 命令还有一点要知道：如果它对主机一无所知，那么它会在终端询问用户。这一行为对 GUI 应用不友好，因此我们会使用 GIT_ASKPASS 环境变量来禁用这个功能。

这个辅助函数的代码如下：

```
GITHUB_HOST = 'github.com'
def git_credentials():
    os.environ['GIT_ASKPASS'] = 'true' ❶
    p = subprocess.Popen(['git', 'credential', 'fill'],
                         stdout=subprocess.PIPE,
```

```
                     stdin=subprocess.PIPE) ❷
stdout,stderr = p.communicate('host={}\n\n'.format(GITHUB_HOST)) ❸

creds = {}
for line in stdout.split('\n')[:-1]: ❹
    k,v = line.split('=')
    creds[k] = v
return creds ❺
```

❶ 我们把 GIT_ASKPASS 设为字符串 'true'。Unix 程序都理解这种做法，这里的作用是让 git-credential 不向用户询问凭据。

❷ 在 Python 中若想使用 stdin 和 stdout，要使用 subprocess.Popen 运行程序。第一个参数是一个列表，列出传给程序的参数，后两个参数指明我们要捕获 stdin 和 stdout。

❸ p.communicate 的作用是写入 stdin，以及返回 stdout 中的内容。这个方法还能返回 stderr 中的内容，不过这个程序将其忽略了。

❹ 处理 stdout 中的内容，在 = 字符处拆分各行，然后存入字典。

❺ 因此，这个函数的返回值是一个字典，存储着 'username' 和 'password' 值。真便利！

4.5.2 窗口和界面

好的，现在我们能跳过登录界面了，但是还没办法向用户显示登录界面。下面实现主窗体，向这个目标靠近。

```
class SearchFrame(wx.Frame):
    def __init__(self, *args, **kwargs): ❶
        kwargs.setdefault('size', (600,500))
        wx.Frame.__init__(self, *args, **kwargs)

        self.credentials = {}
        self.orgs = []

        self.create_controls()
        self.do_layout()
        self.SetTitle('GitHub Issue Search')

        # 先尝试从Git的缓存中加载凭据
        self.credentials = git_credentials()
        if self.test_credentials():
            self.switch_to_search_panel()

        self.Show()
```

❶ 这里有几处句法你可能不理解。*args 和 **kwargs 的作用是捕获多个实参，将其放入一个参数。这里只需知道捕获这些参数的目的是为了传给两行之后的父类构造方法。

__init__ 方法是构造方法，因此主函数调用 SearchFrame() 时从这里开始运行。这个方法的作用概述如下（稍后详解）。

(1) 设置布局尺寸，将其传给父类的构造方法。
(2) 创建 UI 控件。
(3) 使用前面实现的辅助函数获取用户的登录凭据。
(4) 修改标题，向用户显示应用。

在说明这几点的实现方式之前，我们先回过头来说说这个类的作用。这个类负责维护顶层窗体（带有标题栏、菜单等的窗口），决定在窗体中显示什么。这里，我们想先显示登录 UI，获得有效的凭据后（从 Git 的缓存获取，或者由用户输入），切换到搜索 UI。

好了，背景知识够了。下面来看获取和检查凭据的代码。

```
def login_accepted(self, username, password):
    self.credentials['username'] = username
    self.credentials['password'] = password
    if self.test_credentials():
        self.switch_to_search_panel()

def test_credentials(self):
    if any(k not in self.credentials for k in ['username', 'password']):
        return False
    g = Github(self.credentials['username'], self.credentials['password'])
    status,data = g.user.orgs.get() ❶
    if status != 200:
        print('bad credentials in store')
        return False
    self.orgs = [o['login'] for o in data] ❷
    return True

def switch_to_search_panel(self):
    self.login_panel.Destroy()
    self.search_panel = SearchPanel(self,
                                    orgs=self.orgs,
                                    credentials=self.credentials)
    self.sizer.Add(self.search_panel, 1, flag=wx.EXPAND | wx.ALL, border=10)
    self.sizer.Layout()
```

❶ `agithub` 库中的每个函数都返回两个值。在 Python 中可以使用 `a,b = <expr>` 句法直接把两个值绑定给两个变量。

❷ `agithub` 库会把 API 返回的 JSON 解码成 Python 字典。这里，我们只对组织的名称感兴趣，因此我们使用*列表推导*（list comprehension），告诉 Python 只保存 `data` 列表中各个字典里的 `"login"` 字段。

这三个方法在程序执行过程的不同时刻运行。如果凭据从 Git 中读取，立即运行 `test_credentials` 方法；如果在登录面板中输入（参见 4.5.3 节），先运行 `login_accepted` 回调，然后再调用 `test_credentials`。

不管怎么运行，我们的目的都是获取用户所属的组织列表，测试这些方法是否能用。从这

段代码可以看出 agithub 库的使用方式：URL 路径对应于 Github 类实例上的对象 – 属性表示法，HTTP 动词对应于方法调用；返回值是一个字典对象，包含状态码和数据。如果请求失败，即返回的状态码不是 200，向用户显示登录面板。如果成功，调用 switch_to_search_panel 方法。

> 我们在 UI 线程中所做的是同步网络调用。通常这么做不好，因为在处理完网络调用之前 UI 不会有反应。理想的处理方式是，把网络调用移到另一个线程中，得到返回值后发出消息。可是，这样做会增加本章的长度和难度，而本章已经很长很难了，因此没有涉及这些高级话题。希望你能原谅我们做了这样的简化。对这个案例来说，使用同步代码完全没问题。

最后一个方法用于转换 UI。登录面板被两个对象引用，一个是 SearchFrame 实例（父级窗口），一个是控制登录面板布局的布局管理器（sizer）。幸好，调用 Destroy() 方法能去除这两个引用，所以此后可以创建 SearchPanel 实例，再把它添加到布局管理器中。此后还要在布局管理器上调用 Layout() 方法，否则布局管理器不会调整新面板的位置和尺寸。

```
    def create_controls(self):
        # 创建菜单。主要用于在OS X中实现"Cmd-Q"快捷键
        filemenu = wx.Menu()
        filemenu.Append(wx.ID_EXIT, '&Exit')
        menuBar = wx.MenuBar()
        menuBar.Append(filemenu, '&File')
        self.SetMenuBar(menuBar)

        # 实例化登录UI
        self.login_panel = LoginPanel(self, onlogin=self.login_accepted)

    def do_layout(self):
        self.sizer = wx.BoxSizer(wx.VERTICAL)
        self.sizer.Add(self.login_panel, 1, flag=wx.EXPAND | wx.ALL, border=10)
        self.SetSizer(self.sizer)
```

create_controls 方法的作用相当简单，先实例化一个只包含“File → Exit”的菜单，然后实例化登录面板（后文说明实现方式）。注意，创建可见的控件时传给构造方法的第一个参数是 self。这是因为我们构建的 SearchFrame 实例是那个控件的父窗口。

do_layout 方法使用 WxWidgets 提供的布局管理器自动布局。布局管理器是个复杂的话题，不过对于这部分代码来说只需要知道以下几点。

- BoxSizer 在一个方向上堆叠窗体部件，这里是纵向堆叠。
- 传给 sizer.Add 方法的第二个参数是缩放因子。如果设为零，不管父窗口怎么改变尺寸，窗体部件的大小始终不变；如果设为任何非零值，布局管理器控制的所有控件会自动调整，填满整个容器。这个布局管理器中只有一个控件，不过我们仍想让它占满整个窗口，因此传入 1。

- border 参数告诉布局管理器在窗体部件周围留出多少补白空间。
- wx.EXPAND 标志告诉布局管理器，在堆叠的反方向上延伸窗体部件。这里，我们沿纵向堆叠，不过却想横向延伸窗体部件。
- wx.ALL 标志指定在窗体部件的哪边留白。

我们要养成好的习惯，编写测试。能自动验证的代码不多，不过应该覆盖下面这些。

```
from nose.tools import eq_, ok_, raises ❶

class TestApp:
    def setUp(self): ❷
        self.f = None
        self.app = wx.App()

    def tearDown(self):
        if self.f:
           self.f.Destroy()
        self.app.Destroy()

    def test_switching_panels(self): ❸
        self.f = SearchFrame(None, id=-1)
        # 应该出现子面板,而且类型正确
        ok_(isinstance(self.f.login_panel, LoginPanel))
        ok_(isinstance(self.f.search_panel, SearchPanel))
        # 已经销毁
        raises(RuntimeError, lambda: self.f.login_panel.Destroy())
        # 尚未销毁
        ok_(self.f.search_panel.Destroy())
```

❶ 我们使用的测试工具是 Nose，安装方法是执行 pip install nose 命令，运行方法是在命令行中输入 nosetests app.py。这个工具通过命名约定标识测试和固件，用起来基本上很顺手。

❷ Nose 会自动查找 setUp 和 tearDown 方法，分别在运行各个测试之前和之后调用。这里，我们使用那两个方法管理要测试的窗体，以及全部窗体所属的 App 实例。

❸ 这是一个测试方法，Nose 会找到它然后运行。我们确认子面板的类型是否正确，还确认从 Git 中获得凭据后是否会自动过渡到搜索面板。

就这么简单！除了管理几个字段之外，这些代码基本上都用于管理 UI，这正是 UI 类应该做的。下面编写两个面板中的第一个，这个面板会弹入弹出。

4.5.3　登录GitHub

LoginPanel 类的结构与 SearchFrame 类相似，不过有几处明显不同。下面先给出代码，然后再说明。

```
class LoginPanel(wx.Panel):
    def __init__(self, *args, **kwargs):
        self.callback = kwargs.pop('onlogin', None)
        wx.Panel.__init__(self, *args, **kwargs)

        self.create_controls()
        self.do_layout()

    def create_controls(self):
        self.userLabel = wx.StaticText(self, label='Username:')
        self.userBox = wx.TextCtrl(self, style=wx.TE_PROCESS_ENTER)
        self.passLabel = wx.StaticText(self, label='Password (or token):')
        self.passBox = wx.TextCtrl(self, style=wx.TE_PROCESS_ENTER)
        self.login = wx.Button(self, label='Login')
        self.error = wx.StaticText(self, label='')
        self.error.SetForegroundColour((200,0,0))

        # 绑定事件
        self.login.Bind(wx.EVT_BUTTON, self.do_login)
        self.userBox.Bind(wx.EVT_TEXT_ENTER, self.do_login)
        self.passBox.Bind(wx.EVT_TEXT_ENTER, self.do_login)

    def do_layout(self):
        # 使用网格排布控件
        grid = wx.GridBagSizer(3,3)
        grid.Add(self.userLabel, pos=(0,0),
                 flag=wx.TOP | wx.LEFT | wx.BOTTOM, border=5)
        grid.Add(self.userBox, pos=(0,1),
                 flag=wx.EXPAND | wx.LEFT | wx.RIGHT, border=5)
        grid.Add(self.passLabel, pos=(1,0),
                 flag=wx.TOP | wx.LEFT | wx.BOTTOM, border=5)
        grid.Add(self.passBox, pos=(1,1),
                 flag=wx.EXPAND | wx.LEFT | wx.RIGHT, border=5)
        grid.Add(self.login, pos=(2,0), span=(1,2),
                 flag=wx.EXPAND | wx.LEFT | wx.RIGHT, border=5)
        grid.Add(self.error, pos=(3,0), span=(1,2),
                 flag=wx.EXPAND | wx.LEFT | wx.RIGHT, border=5)
        grid.AddGrowableCol(1)

        # 把网格放在纵向中心
        vbox = wx.BoxSizer(wx.VERTICAL)
        vbox.Add((0,0), 1)
        vbox.Add(grid, 0, wx.EXPAND)
        vbox.Add((0,0), 2)
        self.SetSizer(vbox)

    def do_login(self, _):
        u = self.userBox.GetValue()
        p = self.passBox.GetValue()
        g = Github(u, p)
        status,data = g.issues.get()
        if status != 200:
            self.error.SetLabel('ERROR: ' + data['message'])
        elif callable(self.callback):
            self.callback(u, p)
```

其中有些结构与前面的代码类似。先从构造方法说起。

你可能还记得，在 SearchFrame 类的 create_controls 方法中创建这个面板时传入了一个关键字参数：LoginPanel(self, onlogin=self.login_accepted)。定义构造方法时，我们把那个回调取出保存，留待后用。随后，调用两个其他函数，构建布局，然后返回。

create_controls 方法比 SearchFrame 类的版本内容多一些，因为这个面板的控件更多。我们分别使用一行代码创建静态文本、文本输入框和按钮控件。wx.TE_PROCESS_ENTER 样式告诉 WxPython 库，当光标位于文本框内时，如果用户按下回车键，那么就触发一个事件。

接下来的一段代码把控件事件绑定到方法调用上。在 WxPython 中每次触发事件时都会传给处理程序一个参数，这个参数是包含事件相关信息的对象。因此，同一个函数可以处理任何一种事件。这里，两个文本框对 ENTER 事件的处理程序，以及按钮对 BUTTON 事件的处理程序都是 self.do_login。

do_layout 方法使用的布局管理器与之前不同，这里用的是 GridBagSizer 实例。再次重申，布局管理器相关的话题不在本书范畴之内，你只需知道这种布局管理器在网格中排布控件，而且行或列可以拉伸，以填满容器。这里，我们使用 pos=(r,c) 表示法把控件放在各自的位置上（与大多数坐标系不同，这里要先指定所在的“行”），然后通过 span 参数让控件横跨两列。flags 和 border 参数的作用基本与前面一样，AddGrowableCol 函数告知布局引擎允许网格的哪些部分拉伸。

然后我们做了件奇怪的事：把 GridBagSizer 实例放到另一个布局管理器中。嵌套布局管理器是个强大的功能，几乎能实现任何窗口布局，不过实现起来可能不太容易。这个纵向的 BoxSizer 实例也包含几个直接创建的元组，这种特殊的形式用于添加间隔区。这里，我们把包含所有控件的那个布局管理器放在两个比例不同的间隔区之间，距离窗口上部约三分之一处。效果如图 4-5 所示。

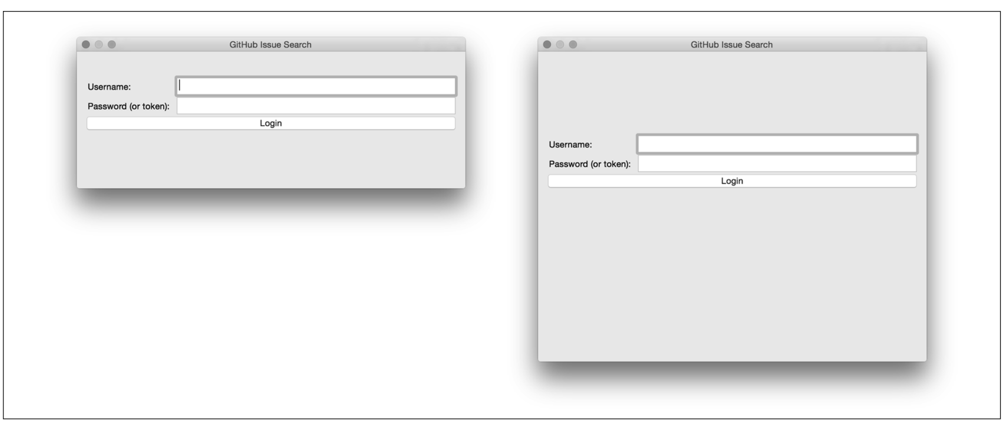


图 4-5：登录 UI 的尺寸变化行为

最后是 do_login 方法。先测试填写的凭据是否正确，如果正确，把凭据传给构造时设置的回调；如果不正确，设置一个标注的文本，这个标注的前景色是带有阴影的红色，起到警醒作用。

下面大致测试这个行为。同样，这个类除了创建 WxPython 控件之外没做什么，不过可以在测试类中加入下述方法，以确认登录失败后会显示错误消息。

```
    def test_login_panel(self):
        self.f = wx.Frame(None)
        lp = LoginPanel(self.f)
        eq_(lp.error.GetLabelText(), '')
        lp.do_login(None)
        ok_(lp.error.GetLabelText().startswith('ERROR'))
```

4.5.4 搜索GitHub

用户成功登录后，销毁 LoginPanel 实例，显示搜索面板（SearchPanel）：

```
class SearchPanel(wx.Panel):
    def __init__(self, *args, **kwargs):
        self.orgs = kwargs.pop('orgs', [])
        self.credentials = kwargs.pop('credentials', {}) ❶
        wx.Panel.__init__(self, *args, **kwargs)

        self.create_controls()
        self.do_layout()

    def create_controls(self):
        self.results_panel = None
        self.orgChoice = wx.Choice(self, choices=self.orgs, style=wx.CB_SORT)
        self.searchTerm = wx.TextCtrl(self, style=wx.TE_PROCESS_ENTER)
        self.searchTerm.SetFocus()
        self.searchButton = wx.Button(self, label="Search")

        # 绑定事件 ❷
        self.searchButton.Bind(wx.EVT_BUTTON, self.do_search)
        self.searchTerm.Bind(wx.EVT_TEXT_ENTER, self.do_search)

    def do_layout(self):
        # 横向排列选择框、搜索框和按钮
        hbox = wx.BoxSizer(wx.HORIZONTAL)
        hbox.Add(self.orgChoice, 0, wx.EXPAND)
        hbox.Add(self.searchTerm, 1, wx.EXPAND | wx.LEFT, 5)
        hbox.Add(self.searchButton, 0, wx.EXPAND | wx.LEFT, 5)

        # 把所有控件移到顶部,留出空间显示结果
        self.vbox = wx.BoxSizer(wx.VERTICAL)
        self.vbox.Add(hbox, 0, wx.EXPAND) ❸
        self.SetSizer(self.vbox)

    def do_search(self, event):
```

```
        term = self.searchTerm.GetValue()
        org = self.orgChoice.GetString(self.orgChoice.GetCurrentSelection())
        g = Github(self.credentials['username'], self.credentials['password'])
        code,data = g.search.issues.get(q="user:{} {}".format(org, term)) ❹
        if code != 200:
            self.display_error(code, data)
        else:
            self.display_results(data['items'])

    def display_results(self, results): ❺
        if self.results_panel:
            self.results_panel.Destroy()
        self.results_panel = SearchResultsPanel(self, -1, results=results)
        self.vbox.Add(self.results_panel, 1, wx.EXPAND | wx.TOP, 5)
        self.vbox.Layout()

    def display_error(self, code, data): ❻
        if self.results_panel:
            self.results_panel.Destroy()
        if 'errors' in data:
            str = ''.join('\n\n{}'.format(e['message']) for e in data['errors'])
        else:
            str = data['message']
        self.results_panel = wx.StaticText(self, label=str)
        self.results_panel.SetForegroundColour((200,0,0))
        self.vbox.Add(self.results_panel, 1, wx.EXPAND | wx.TOP, 5)
        self.vbox.Layout()
        width = self.results_panel.GetSize().x
        self.results_panel.Wrap(width)
```

这个类的代码有点多，不过有些是见过的。我们不一一详述，下面介绍几个有趣的功能。

❶ 创建这个面板时，我们通过关键字参数传入用户的凭据和组织列表，因此这些信息在 `kwargs` 字典中。这里，我们使用 `pop` 方法取出父类的构造方法不理解的值。

❷ 既捕获搜索按钮的“点击”事件，又捕获文本框的“回车键”事件。二者都能触发搜索。

❸ 把搜索栏添加到布局管理器中时，我们把缩放因子设为 `0`。因此，有多余空间时搜索框不会延伸，而是保持尺寸不变，为后面添加的结果面板留出空间。

❹ 执行搜索操作。首先获取搜索词条和组织，然后将其传给 `agithub` 实例，返回结果和 HTTP 状态码。

❺ 把搜索结果传给另一个类，然后把那个类的实例添加到主布局管理器中，并且在参数中指明，填满余下的可用空间。

❻ 如果搜索出错，那么使用这个方法把错误显示出来。其中有几行代码根据搜索面板的既定宽度调整文本的换行宽度。这种方式不是很好，更好的处理方式留作练习，由读者完成。

同样，代码虽多，但是大多数都见过。覆盖上述代码的测试代码如下：

```
    def test_search_panel(self):
        self.f = wx.Frame(None)
        sp = SearchPanel(self.f, orgs=['a', 'b', 'c'])
        eq_(0, sp.orgChoice.GetCurrentSelection())
        eq_('a', sp.orgChoice.GetString(0))
        sp.display_error(400, {'errors': [{'message': 'xyz'}]})
        ok_(isinstance(sp.results_panel, wx.StaticText))
        eq_('xyz', sp.results_panel.GetLabelText().strip())
```

4.5.5 显示结果

至此，登录面板有了，用户也能输入查询词条了，可是还不能显示结果。下面就来解决这个问题。

获得搜索结果后，先创建一个 SearchResultsPanel 实例，然后再创建一系列 SearchResult 实例。下面介绍一下这两个类。

```
class SearchResultsPanel(wx.ScrolledWindow): ❶
    def __init__(self, *args, **kwargs):
        results = kwargs.pop('results', [])
        wx.PyScrolledWindow.__init__(self, *args, **kwargs)

        # 把搜索结果控件布局到可滚动的区域中
        vbox = wx.BoxSizer(wx.VERTICAL)
        if not results:
            vbox.Add(wx.StaticText(self, label="(no results)"), 0, wx.EXPAND)
        for r in results:
            vbox.Add(SearchResult(self, result=r),
                     flag=wx.TOP | wx.BOTTOM, border=8)
        self.SetSizer(vbox)
        self.SetScrollbars(0, 4, 0, 0)

class SearchResult(wx.Panel):
    def __init__(self, *args, **kwargs):
        self.result = kwargs.pop('result', {})
        wx.Panel.__init__(self, *args, **kwargs)

        self.create_controls()
        self.do_layout()

    def create_controls(self): ❷
        titlestr = self.result['title']
        if self.result['state'] != 'open':
            titlestr += ' ({})'.format(self.result['state'])
        textstr = self.first_line(self.result['body'])
        self.title = wx.StaticText(self, label=titlestr)
        self.text = wx.StaticText(self, label=textstr)

        # 调整标题的文本格式
        titleFont = wx.Font(16, wx.FONTFAMILY_DEFAULT,
                            wx.FONTSTYLE_NORMAL, wx.FONTWEIGHT_BOLD)
        self.title.SetFont(titleFont)
```

```
        # 在这整个控件上绑定点击和悬停事件 ❸
        self.Bind(wx.EVT_LEFT_UP, self.on_click)
        self.Bind(wx.EVT_ENTER_WINDOW, self.enter)
        self.Bind(wx.EVT_LEAVE_WINDOW, self.leave)

    def do_layout(self):
        vbox = wx.BoxSizer(wx.VERTICAL)
        vbox.Add(self.title, flag=wx.EXPAND | wx.BOTTOM, border=2)
        vbox.Add(self.text, flag=wx.EXPAND)
        self.SetSizer(vbox)

    def enter(self, _):
        self.title.SetForegroundColour(wx.BLUE)
        self.text.SetForegroundColour(wx.BLUE)

    def leave(self, _):
        self.title.SetForegroundColour(wx.BLACK)
        self.text.SetForegroundColour(wx.BLACK)

    def on_click(self, event): ❹
        import webbrowser
        webbrowser.open(self.result['html_url'])

    def first_line(self, body):
        return body.split('\n')[0].strip() or '(no body)'
```

❶ 容器面板特别简单，只有一个构造方法。这个类的作用是盛放结果，在一个可以滚动的窗口中呈现出来。

❷ `SearchResult` 实例有两个静态文本控件组成，分别显示工单的标题和正文的第一行。

❸ 我们不仅为这整个面板绑定了点击事件，还绑定了鼠标移入和移出事件，因此其行为与浏览器中的链接很像。

❹ 这是在 Python 中使用默认浏览器打开 URL 的方式。

以上就是一个简单的 WxPython 应用的代码。使用这个库要按照特定的风格编写代码，有点繁琐。不过好的一面是，这里毫无隐藏，应用的所有布局都通过代码实现，没有看不见的戏法。此外，使用这个库构建的应用无需修改就能在任何一台电脑中运行，这是一个巨大的优势。WxPython 可能缺少新兴框架的某些功能，可是它能快速构建简单的跨平台 UI，这才是重点。

代码就分析到这里。如果你一直跟着我做，把所有代码都输入到一个文件中了，那么现在可以运行那个文件，搜索工单了。然而，这个应用是针对非技术类用户的。下面看看我们能做些什么，让这类用户更易于上手使用。

4.6 打包

我们不想让任何用户安装 Python 2.7 和一堆包，因此可以使用 PyInstaller 打包应用，得到

易于分发和运行的文件。

假设你把前面的代码都写入 search.py 文件了，而且 agithub.py 文件在同一个目录中。让 PyInstaller 打包应用的方式如下：

```
$ pyinstaller -w search.py
```

就这么简单！ `-w` 标志的作用是让 PyInstaller 创建带窗口的应用，而不是默认的控制台形式。在 OS X 中，上述命令会生成 search.app 应用包，在 Windows 中则生成 search.exe 文件。你可以把这两个文件复制到没有安装 Python 的电脑中，它们都能正常运行。

这是因为 PyInstaller 复制了运行程序所需的一切，包括 Python 解析器和代码文件本身。执行上述命令生成的文件大小为 67MB，对这么简单的程序来说似乎有点大，但是考虑到打包了如此多的内容，这个大小还算说得过去。

4.7　小结

啊，这一章的内容真多呀！现在休息一下，回顾所学的知识。

本章给出的代码主要用于定义图形界面。这种任务所需的代码都十分繁琐，因为这类任务特别复杂。可是，有了 WxPython 之后，我们可以使用 Python 编写 GUI 应用，写起来不比其他工具包难，而且应用能在所有主流平台中免费运行。

我们学习了如何使用 `git credential` 命令向 Git 询问某个 Git 服务器的凭据。这个功能特别强大，除此之外还能定制凭据存储后台，不过我们只了解了皮毛。借助这个功能，我们可以顺应用户现有的习惯，不再一次次向用户询问相同的信息。

我们还学习了如何使用 `agithub` 库优雅地抽象 HTTP API。我们使用对象 – 方法表示法请求搜索工单的 API，进行身份验证和工单搜索。`agithub` 库充分考虑了向前兼容性和惯用法，它把属性和方法串接起来构建查询 URL。使用这种方式也能查询其他 REST API。

其实，本章的要点是学习如何使用 GitHub Search API。我们学习了这个 API 的一般行为、不同的搜索类型、解释和排序结果的方式，以及如何调整搜索词条，减少无关结果的数量。掌握这些知识后，你将能在 GitHub 或 GitHub Enterprise 中找到任何想找的信息。我们还了解到，GitHub 网站中的搜索 UI 是对搜索 API 的简单包装，所以不管自己编写代码还是直接在浏览器中搜索，都能使用相同的技巧和技术。

该换个话题了。下一章介绍 Commit Status API，通过这个 API 能为 Git 仓库里的提交添加注解，标上“好”或“坏”标志。下一章中将使用几年前盛极一时的技术方案：C# 和 CLR。

第 5 章

.NET和Commit Status API

往简单了说，Git 仓库就是一长串提交，每个提交都包含相当多的信息：源码文件的内容、提交人和提交时间，以及提交人对改动的说明，等等。这些都是有用的信息，体现了 Git 的主要用途，即管理软件项目的历史。

GitHub Commit Status API 为提交添加了额外的元数据：提交的各种服务状态。这个功能主要把信息显示在拉取请求界面中，如图 5-1 所示。拉取请求中的每个提交都有符号标明的状态，红色“×”表示失败或错误，绿色“✓”表示成功，黄褐色“•”表示正在判定中。这个功能在拉取请求的底部也有体现，如果分支中最后一个提交没有被标记为成功，那么你会看到警告，提醒你不要合并请求。

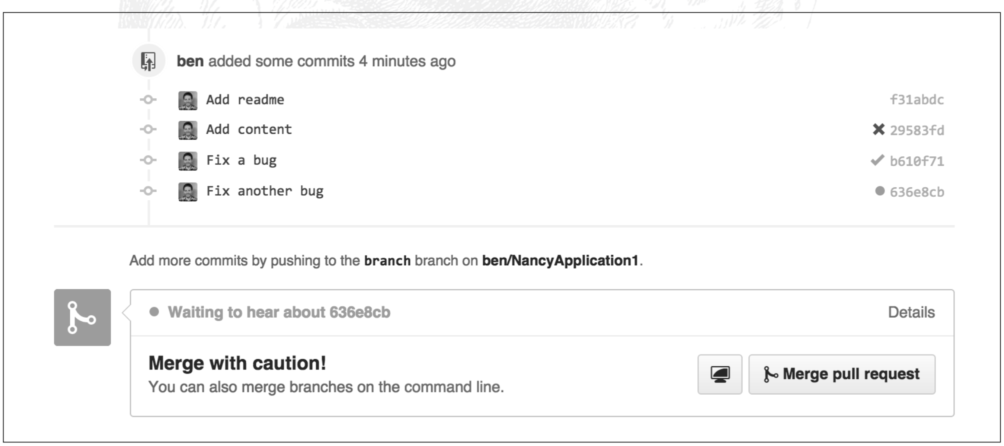


图 5-1：拉取请求界面显示的提交状态

显然，这个功能对持续集成服务最有用。向一个分支中推送新提交后，Jenkins 这样的程序会收到通知，然后用新代码执行一次构建 / 测试过程，最后通过提交状态 API 报告结果。这类应用甚至还可以提供一个链接，指向构建结果，让用户查明哪些测试失败了。这个功能汇聚了判断是否接受提案所需的各种信息，包括改动了哪些代码、人们的看法如何，以及改动是否破坏了什么。这些问题的答案可在同一页面中找到，即拉取请求讨论页面。

不过，构建和测试只是开端，提交的状态还有其他用途。例如，开源项目通常都有授权协议，如果想提交贡献，必须签署这个协议。这叫“贡献者授权协议”（Contributor License Agreements，CLA），通常会要求你同意把自己的贡献授权给项目的维护者。可是，手工检查每个拉取请求，确认作者是否签署贡献者授权协议太麻烦，此时可以使用类似持续集成的服务。CLAHub 就是这样的服务，它会检查是否提交的所有作者都签署了贡献者授权协议，如果没有，那么把最后一个提交标记为“错误”。

现在，我们知道这个功能的目的和作用了，下面来看如何使用程序与它交互。

5.1 Commit Status API

首先讨论访问控制。Commit Status API 对 OAuth 的需求比其他 API 强烈。把仓库设为私有后，我们能完全控制哪些人或哪些应用可以访问仓库。我们会不假思索地信任 GitHub 的内部代码，相信它能保全我们的数据，可是互联网上的应用呢？借助 OAuth，我们可以为应用赋予访问私有仓库的权限，不过是带有*限制*的，应用要使用 OAuth 的作用域征询特定的权限，不能对数据肆意妄为。而且，这样还能完全掌控各种权限，随时都能取消应用的访问权。

OAuth 系统使用作用域管理权限，应用发出请求，被授予特定的作用域之后才能执行特定的操作。访问 Commit Status API 需要 `repo:status` 这个 OAuth 作用域，授予之后，应用可以读写提交状态，不过*仅限于此*，应用无法访问仓库中的内容。这似乎有点奇怪，不审查内容怎么判断提交的状态呢？要知道，这个功能除了持续集成之外还有其他使用场景，也就是说应用做判断时可能不需要全部访问权限。如果服务需要查看仓库的内容，可以请求 `repo` 作用域，获得对整个仓库内容（包括提交状态）的读写权限。写作本书时还没办法只请求仓库的读权限，因此如果一项服务需要访问你的数据，你不得不信任它，为它开放写权限。

此外，这个 API 也可以匿名访问，完全不使用 OAuth。不过，此时只能读取公开仓库的状态，没有写权限，而且不能访问私有仓库。

下表对以上内容做了总结。

OAuth作用域	状态的访问权限	仓库数据的访问权限
无（匿名）	只能读公开仓库	只能读公开仓库
repo:status	读 / 写	无
repo	读 / 写	读 / 写

5.1.1　原始状态

现在我们知道如何获得访问提交状态的权限了，下面来看提交状态包含什么内容。提交状态是原子实体，每个提交几乎可以有任意多个状态（数以千计）。查询现有状态的方法是，向 API 服务器中的 /repos/<*user*>/<*repo*>/<*ref*>/statuses 端点发送 GET 请求。得到的响应是状态列表，如下所示。

```
[
  {
    "url": "https://api.github.com/repos/…",
    "id": 224503786,
    "state": "success",
    "description": "The Travis CI build passed",
    "target_url": "https://travis-ci.org/libgit2/libgit2/builds/63428108",
    "context": "continuous-integration/travis-ci/push",
    "created_at": "2015-05-21T03:11:02Z",
    "updated_at": "2015-05-21T03:11:02Z"
  },
  …
]
```

其中大多数字段不言自明，不过有几个需要解析一下。

字　段	说　明
state	值为 success、failure、error 或 pending
target_url	为提交做出某个判断的 URL（这里指向的是构建 / 测试日志），帮助用户查明判断依据
context	用于把同一个服务的多个状态更新关联起来；各个服务根据自己的规则设置这个字段，不过对于任何输出状态的过程来说，输出 pending 状态和最终状态时要使用相同的值

这个 API 用于获取原始数据，不过很快就会变得复杂。如何判断提交的状态是“好的”呢？如果有多个状态且依次为三个待定状态、一个成功状态、一个待定状态、两个失败状态和另一个成功状态，这时该怎么判断呢？我们可以通过 context 字段把同一个服务的状态关联起来，然后按照 created_at 字段排序，查看各个阶段的状态，可是这样工作量太大。幸好，GitHub API 服务器可以代劳。

5.1.2　合并后的状态

如果向 /repos/<*user*>/<*repo*>/<*ref*>/status（注意最后一个单词是单数）发送 GET 请求，那么获得的响应是下面这样的。

```
{
  "state": "success",
  "statuses": [
    {
      "url": "https://api.github.com/repos/…",
      …
    },
    { … }
  ],
  "sha": "6675aaba883952a1c1b28390866301ee5c281d37",
  "total_count": 2,
  "repository": { … },
  "commit_url": "https://api.github.com/repos/…",
  "url": "https://api.github.com/repos/…"
}
```

`statuses` 字段中的数组是需要你自己编写代码实现特定的逻辑得到的结果，它按照上下文合并状态，只保留最后一个状态。`state` 字段的值是综合所有上下文，根据下述规则得出的最终状态。

状　　态	致　　因
`failure`	任一上下文输出 `failure` 或 `error` 状态
`pending`	某个上下文的最终状态是 `pending`，或者没有状态
`success`	每个上下文的最终状态都是 `success`

这可能正是你所需要的，不过如果你的使用场景需要其他规则，完全可以使用 `statuses` 端点获取原始数据，然后自己合并状态。

5.1.3　创建状态

显然，目前提到的状态都从别处而来。这个 API 还支持创建状态，方法是向 /repos/<*user*>/<*repo*>/statuses/<*sha*> 发送 POST 请求，并在请求中发送一个 JSON 对象，提供想包含在状态中的字段。

字　　段	说　　明
`state`	必须是 `pending`、`success`、`error` 或 `failure`（必需）
`target_url`	指向判定状态详情的链接
`description`	一段简短的文字，说明为了判定状态，服务做了什么
`context`	各个应用专用的字符串，以便 API 管理多个服务为同一个提交判定的状态

注意，该 URL 的最后一部分是 <*sha*>。查询状态或合并的状态使用的是引用名（如 `master`），而创建状态时要知道提交的完整 SHA-1 散列值，这样才能注解那个提交。这是为了避免条件竞争，如果使用引用，处理开始和结束时指向的可能不是同一个提交，而提交的 SHA 永远不变。

5.2 编写一个应用

好了，现在我们知道如何读写状态了，下面来运用这个 API。这一节我们将构建一个简单的 HTTP 服务，通过 OAuth Web 流程获得授权后，为有权限访问的仓库创建提交状态。我们将构建的这个系统作用有限，却是以后根据特定需求定制开发的良好起点。

这一次我们使用的语言是 C#，运行在 CLR（Common Language Runtime，公共语言运行时）平台上。在信息技术历史上的某段时期，对本书而言这不是好的选择，因为 CLR 只能在 Windows 中使用，开发工具价格不菲，而且 C# 语言和库有相当大的局限性。然而，Mono（.NET 运行环境的开源实现）出现了，这是 CLR 核心的开源版，而且提供了免费工具，因此 C# 现在已被广泛接受，对开源和业余开发者来说是相当不错的技术选择。此外，Mono 的包生态系统充满生机，我们可以利用现有的库简化我们的工作。

5.2.1 要使用的库

本章不会从头动手编写整个 HTTP 服务器。听到这句话你一定很高兴。有很多开源的包能为我们代劳，这个项目要使用的是 Nancy。起初，Nancy 项目是作为 Ruby 中 Sinatra 框架的 CLR 端口起步的（项目名称取自 Frank Sinatra 的女儿 Nancy）。你会发现这个库的功能非常强大，同时特别简洁。

我们也不会直接实现访问 GitHub API 的代码，因为 GitHub 提供了 CLR 库，名为 octokit.net。这个库能正确处理一切操作，而且考虑到了异步和类型安全。GitHub 为 Windows 系统开发的客户端使用的也是这个库，所以定能胜任这个小应用的需求。不过，这个库对项目的搭建有特殊的要求：为了正常运行，必须使用新版 CLR（4.5）。如果想知道如何避免受此限制，想跟着我一起做，请继续阅读下一节。如果你以前用过 Nancy，而且已经安装了 NuGet 包，基本上可以跳到 5.2.3 节。

5.2.2 开发环境

如果你想跟着我一起编写代码，请参照下文搭建开发环境，安装所需的全部工具。搭建的过程在 Windows（使用 Visual Studio）和其他平台（使用 Xamarin 工具）中有所不同。

1. Visual Studio

如果你使用的是 Windows 系统，访问 https://www.visualstudio.com/，下载 Visual Studio 社区版。安装程序会给出众多选项，而这个示例只需要“Web 开发者”组件，不过你可以根据自己的需求勾选其他组件。（如果你购买了 Visual Studio 的高级版本，或者已经安装了 Web 开发相关的包，那就一切就绪了。）

为了让开发更顺利，我们要安装一个插件——Nancy 项目模板。访问 https://visualstudiogallery.msdn.microsoft.com/，然后搜索“nancy.templates”。在搜索结果中选择

属于 NancyFx 组织的“Nancy.Templates”，然后点击“Get Now”。此时会下载一个 .vsix 文件，下载完成后双击，把模板安装到 Visual Studio 中。[1]

下一步，使用刚才安装的模板新建一个项目。进入“File → New Project”菜单，然后在模板列表中选择“Visual C# → Web → Nancy Application with ASP.NET Hosting”（如图 5-2 所示）。根据需求调整底部的路径和名称，然后点击 OK。

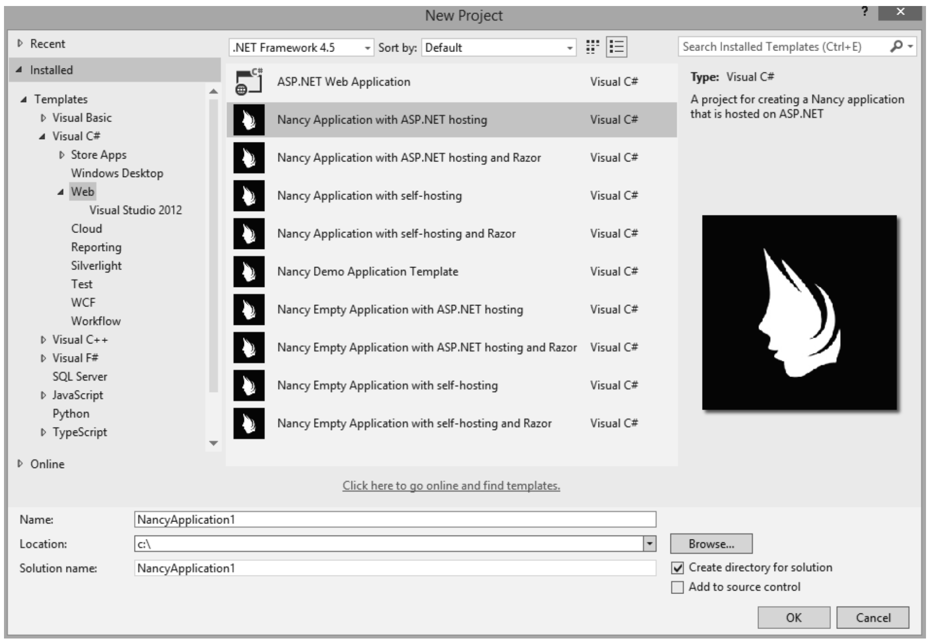


图 5-2：在 Visual Studio 中创建一个 Nancy 应用

接下来，把目标 CLR 框架改为 Octokit 支持的版本。在 Solution Explorer（解决方案资源管理器）中右键点击项目节点，然后选择“Properties”。在 Application 选项卡中，把 Target Framework 设为 .NET 4.5（或以上版本），然后保存。此后，可能会要求你重新加载这个解决方案。

最后一步，使用 NuGet 安装 Nancy 和 Octokit 两个包。在解决方案资源管理器中右键点击项目节点，然后选择“Manage NuGet Packages”。搜索“Nancy”，如果需要，升级，因为 Nancy 项目模板指定的版本可能过时了。然后，搜索“Octokit”，安装。至此，我们得到了一个空的解决方案，而且已经配置好，可以编写代码了。如果想启动调试功能，进入“Debug → Start Debugging”菜单，或者按 F5 键。Visual Studio 会在调试器中启动服务器，

注 1：这个 .vsix 文件不支持新版 Visual Studio，如果安装出错，可以从这里下载：https://github.com/nerobianchi/visual_studio_gallery。——译者注

在 IE 中打开 http://localhost:12008/（端口可能不同），显示 Nancy 默认的“404 Not Found”页面。

2. Xamarin Studio

写作本书时，在 OS X 和 Linux 中最简单的搭建方法是，访问 http://www.monodevelop.com/，下载和安装 MonoDevelop。Mono 是微软 CLR 规范的开源实现，MonoDevelop 是与 Visual Studio 类似的开发环境，不过是使用 Mono 构建的，而且完全开源。如果在 Windows 或 OS X 设备中安装 MonoDevelop，它会提醒你改为安装 Xamarin Studio。这是 MonoDevelop 的新版，功能更多，也能正常运行本章的示例。

因为这两个 IDE 无需安装针对 Nancy 的项目模板，所以直接新建一个空 Web 项目即可。进入“File → New → Solution”菜单，在模板选择对话框中选择“ASP.NET → Empty ASP.NET Project”，如图 5-3 所示。

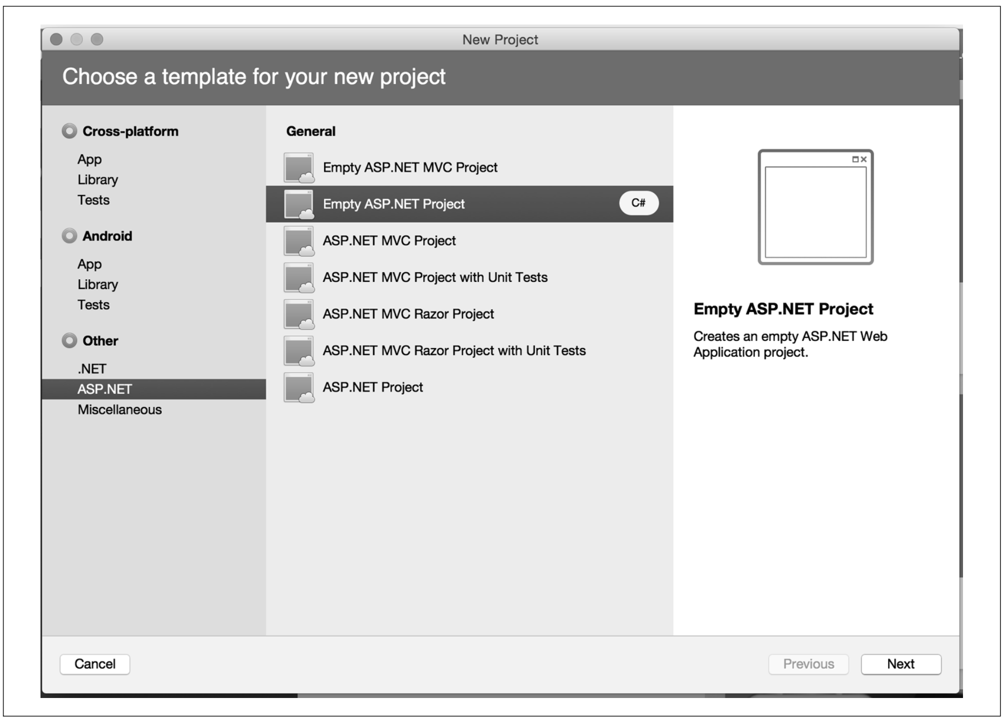


图 5-3：在 Xamarin Studio 中创建一个空 ASP.NET 项目

这个向导接下来的步骤用于设置项目名称和路径，根据需求填写即可。

接下来，更新目标框架设置。在项目（不是解决方案）对应的解决方案资源管理器中，按住 Control 键点击或者右键点击节点，从弹出的菜单中选择 Options（选项）。在“Build → General”选项卡中，把 Target Framework 设为“Mono / .NET 4.5”（或以上版

本），然后点击 OK。

最后，使用 NuGet 安装 Nancy 和 Octokit 两个包。进入“Project → Add NuGet Packages”菜单，打开包管理器。搜索 Nancy，勾选旁边的选择框，再搜索 Octokit，勾选旁边的选择框，然后点击右下部的“Add Packages”。安装好之后，本章示例代码所需的环境就搭建好了。如果想启动调试器，进入“Run → Start Debugging”菜单，或者按 ⌘-Enter 键。Xamarin 会启动服务器，并在浏览器中打开 http://127.0.0.1:80080（端口可能不同），现在浏览器中会显示默认的“404 Not Found”页面。

5.2.3 发送请求

好了，项目创建好了，该编写代码了。下面我们要让 Nancy 应用运行起来。作为优秀的工程师，我们要先编写测试。为此，要为现有的应用项目生成一个单元测试项目，然后添加一个 NuGet 引用，指向 `Nancy.Testing` 包。此后，你可以复制测试示例，粘贴到模板生成的默认测试模块顶部。

我们将首先编写通过一个端点获取用户粉丝数量的代码。为了对此进行测试，我们选择一个有名的用户，确认能获取他的真名。测试代码如下。

```
using NUnit.Framework;
using Nancy;
using Nancy.Testing;
using Nancy.Bootstrapper;
using System.Collections.Generic;
using Nancy.Session;

namespace NancyApplication1.Tests
{
  [TestFixture ()]
  public class Test
  {
    private Browser browser;

    [SetUp]
    public void Setup(){
      this.bootstrapper =
        new ConfigurableBootstrapper(with => {
        with.Module<Handler>();
      });
      this.browser = new Browser (bootstrapper);
    }

    [Test ()]
    public void FetchesUserDetails ()
    {
      var result = this.browser.Get ("/mojombo", ❶
        with => with.HttpRequest ());
      Assert.AreEqual (HttpStatusCode.OK, result.StatusCode);
```

```
            Assert.IsTrue (result.Body.AsString()
              .Contains("Tom Preston-Werner")); ❷
        }
      }
    }
```

❶ 这里使用 Nancy.Testing 包提供的 Browser 类向 /mojombo 发起请求，得到的响应应该是该用户的粉丝数。

❷ 这里断言从端点中获得了 mojombo 的真名。

失败的测试有了，下面编写代码，使用 Nancy 实现该端点。这部分代码的初始版本如下。

```
using Nancy;
using Octokit;
using System;
using System.Collections.Generic;
using System.Linq;

namespace NancyApp
{
    public class Handler : NancyModule ❶
    {
        private readonly GitHubClient client =
            new GitHubClient(new ProductHeaderValue("MyHello")); ❷

        public Handler()
        {
            Get["/{user}", true] = async (parms, ct) => ❸
                {
                    var user = await client.User.Get(parms.user.ToString()); ❹
                    return String.Format("{0} people love {1}!",
                                         user.Followers, user.Name); ❺
                };
        }
    }
}
```

❶ 定义 NancyModule 类的子类，在 Nancy 中接收和处理 HTTP 请求只需这样做即可。

❷ GitHubClient 类是 Octokit 的入口点。我们创建一个实例留待后用，这里把产品名设为一个占位符，我们访问的这个 API 用不到这个名称。

❸ 这个模块的构造方法要求提供路由映射。我们使用 NancyModule 中的 Get 字典把 /{user} 映射到一个匿名函数上。索引运算符的第二个参数表明处理函数是异步的。

❹ 这一行展示如何从请求 URL 中获取 {user} 部分（是 parms 参数的一个属性），以及如何使用 Octokit 查询 GitHub User API。注意，要使用 await 等待网络查询返回结果，因为查询可能要花点时间。

❺ Nancy 处理请求的函数可以直接返回文本字符串，浏览器会把这个字符串视为 HTML 代码。这里，我们返回一个简单的字符串，包含用户名和粉丝数。

> 特别留意 `async` 和 `await` 关键字。这两个关键字构成一种特殊的句法，把在一个事件循环中运行的一系列函数封装起来。看起来这些代码是依序运行的，但是遇到 `await` 关键字时，系统会发起异步请求，然后把控制权交还主事件循环。请求结束后，即实现承诺后，事件循环会回到期待这个 `await` 关键字的返回值那里，此时作用域中的所有变量都完好无损。这个功能在 .NET 4.0（2012 年发布）中引入，能以几乎与同步代码一样的方式编写异步代码。这是众多开发者青睐 C# 的原因之一。

与“hello, world”相比，这个示例有些复杂，不过还算简洁明了。这是个好的开头，下面实现 OAuth 验证时还要复杂一些。

5.2.4 OAuth验证流程

更新提交状态之前，要获得用户的授权。除了询问用户的用户名和密码之外（这种方式获得的权限太大，如果启用了双重身份验证，权限还不够用），只能使用 OAuth，而这种验证方式并不那么简单。

下面以这个简单的服务器为例概述 OAuth 验证流程。

(1) 如果没有授权令牌，或者令牌已过期，要请求授权令牌。令牌就是一个字符串，不过不能由我们自己生成，而要向 GitHub 索要。方法是，重定向用户的浏览器，转到一个 GitHub API 端点，在查询参数中发送所需的权限类型和其他细节。
(2) GitHub（在浏览器中）告知用户有应用在请求权限，用户可以同意，也可以拒绝。
(3) 如果用户同意授权，浏览器会重定向到第一步指定的 URL，并在查询参数中传送一个“代码”。这不是我们所需的访问令牌，而是获得令牌的时限密钥。
(4) 在处理请求的函数中，可以使用一个 REST API 获取真正的 OAuth 访问令牌，然后将其存储在安全的地方。之所以这么做是因为，如果有令牌就可以跳过前面几步。
(5) 现在有权限了，可以在已验证模式中使用 GitHub API。

过程看起来有点复杂，不过这样设计有几个好处。首先，可以按作用域划分权限，很少有应用需要访问用户的账户和数据的全部权限。其次，整个交互过程是安全的，至少其中有一步要用户手动操作，无法自动完成。最后，访问令牌绝不会传到用户的浏览器中，这样规避了一整类安全漏洞。

下面分析我们这个简单服务器实现 OAuth 验证流程的代码。首先，如果已经获得令牌，我们把它存储起来，这样每次请求时就不用走一遍整个重定向循环了。我们会把令牌存在 cookie 中（此时令牌会在用户的浏览器中传来传去，在生产环境中，应用或许会把令牌存入数据库）。Nancy 能帮助我们完成这项操作，不过在此之前要启用 Nancy，而启用的方法是使用启动加载器（bootstrapper）。在应用中添加下面这个类：

```
using Nancy;
using Nancy.Bootstrapper;
using Nancy.Session;
using Nancy.TinyIoc;

namespace NancyApp
{
    public class Bootstrapper : DefaultNancyBootstrapper
    {
        protected override void ApplicationStartup(TinyIoCContainer container,
                                                   IPipelines pipelines)
        {
            CookieBasedSessions.Enable(pipelines);
        }
    }
}
```

Nancy 能自动检测到启动加载器类，然后使用它初始化服务器。现在，在 `NancyModule` 中可以使用 `Session` 属性存取 cookie 中的值。

接下来，因为某些请求要使用应用的 ID 和密令，所以要把这两个值写入代码。方法是，在 `Handler` 类中添加下述两个字段。如果没有应用，访问 https://github.com/settings/developers 创建一个，把回调 URL 设为 `http://localhost:8080/authorize`（在不同的环境中，端口号可能稍有不同），稍后解释为什么这么做。

```
        private const string clientId = "<clientId>";
        private const string clientSecret = "<clientSecret>";
```

显然，如果你一直跟着做，那么你应该使用自己的 API 应用的 ID 和密令。

接下来需要一个辅助方法，让它启动 OAuth 验证流程。

```
private Response RedirectToOAuth()
{
    var csrf = Guid.NewGuid().ToString();
    Session["CSRF:State"] = csrf; ❶
    Session["OrigUrl"] = this.Request.Path; ❷

    var request = new OauthLoginRequest(clientId)
        {
            Scopes = { "repo:status" }, ❸
            State = csrf,
        };
    var oauthLoginUrl = client.Oauth.GetGitHubLoginUrl(request);
    return Response.AsRedirect(oauthLoginUrl.ToString()); ❹
}
```

❶ CSRF 是跨站请求伪造（cross-site request forgery）的简称。这个机制能确保 OAuth 请求确实是由我们的网站发出的。用户授权访问后，GitHub OAuth API 会传回这个值，我们把它存入 cookie，以备后用。

❷ 把源 URL 存入会话 cookie 是基于用户体验方面的考虑，因为 OAuth 验证流程结束后，我们想把用户带回他们一开始想执行操作的页面。

❸ `repo:status` 是我们请求的权限。注意，这个对象中还包括 CSRF 令牌，这样 GitHub 才会把它传回给我们，用于后面的验证。

❹ 这里使用 Octokit 生成重定向 URL，然后把用户的浏览器转到该地址。

如果发现令牌缺失或无效，那么这个模块中的任何一个路由处理函数都能调用 `RedirectToOAuth` 方法。稍后会说明如何调用，现在先继续处理 OAuth 验证流程。

在 GitHub 中设置应用时，我们指定了一个授权回调 URL。这里，我们指定的是 `http://localhost:8080/authorize`。如果 GitHub 决定向应用授权所请求的权限，它会把用户的浏览器重定向到该 URL。那个端点的处理函数如下，这段代码要写入模块的构造方法中。

```
Get["/authorize", true] = async (parms, ct) => ❶
    {
        var csrf = Session["CSRF:State"] as string;
        Session.Delete("CSRF:State");
        if (csrf != Request.Query["state"]) ❷
        {
            return HttpStatusCode.Unauthorized;
        }

        var queryCode = Request.Query["code"].ToString();
        var tokenReq =  new OauthTokenRequest(clientId, ❸
                                              clientSecret,
                                              queryCode);
        var token = await client.Oauth.CreateAccessToken(tokenReq);
        Session["accessToken"] = token.AccessToken; ❹

        var origUrl = Session["OrigUrl"].ToString();
        Session.Delete("OrigUrl");
        return Response.AsRedirect(origUrl); ❺
    };
```

❶ 使用 Nancy 要像这样把路径映射到处理函数上。`NancyModule` 的子类都有一个可索引对象，用于处理各个 HTTP 动词，而且各个动词都可以依附同步或异步处理函数。此外，还可以包含 URL 中动态变化的部分，稍后会见到一例。

❷ 这行代码验证之前生成的 CSRF 令牌。如果不匹配，说明有可疑的事情，因此返回 401 响应码。

❸ 请求 REST API，把 OAuth 代码转换成访问令牌。为了保证的确是我们的应用在请求令牌，我们传入了客户端 ID 和密令，以及 GitHub 提供给我们的代码。

❹ 这行代码把得到的令牌存入会话 cookie。再次声明，真正的应用不该这么做，但是对这个示例应用而言无妨。

❺ 这行代码把用户重定向到最初想执行操作的页面，尽量不打断原来要做的事情。

这个端点可以测试，不过要想办法处理会话。为此，我们把下述代码片段添加到测试项目的命名空间中。

```
public static class BootstrapperExtensions
{
  public static void WithSession(this IPipelines pipeline,
                                     IDictionary<string, object> session)
  {
    pipeline.BeforeRequest.AddItemToEndOfPipeline(ctx =>
      {
        ctx.Request.Session = new Session(session);
        return null;
      });
  }
}
```

这是个*扩展方法*（extension method），作用是为请求提供 `Session` 对象，用于处理 CSRF。定义好这个方法之后，在测试组件类中添加一个测试方法：

```
[Test]
public void HandlesAuthorization()
{
  // CSRF令牌不匹配
  bootstrapper.WithSession(new Dictionary<string, object> {
    { "CSRF:State", "sometoken" },
  });
  var result = this.browser.Get ("/authorize", (with) => {
    with.HttpRequest();
    with.Query("state", "someothertoken");
  });
  Assert.AreEqual (HttpStatusCode.Unauthorized, result.StatusCode);

  // CSRF令牌匹配
  bootstrapper.WithSession(new Dictionary<string, object> {
    { "CSRF:State", "sometoken" },
              { "OrigUrl", "http://success" },
  });
  result = this.browser.Get ("/authorize", (with) => {
    with.HttpRequest();
    with.Query("state", "sometoken");
  });
  result.ShouldHaveRedirectedTo ("http://success");
}
```

第一部分设置一个不匹配的 CSRF 令牌。会话中的令牌是 `"sometoken"`（在请求 API 之前设定），而请求中的令牌是 `"someothertoken"`（应该由 GitHub 发送），所以我们断言状态码是 401。第二部分用的是匹配的令牌，所以我们断言响应是重定向，转到会话中存储的 URL。

做完这些之后，我们便得到了令牌，接下来可以愉快地继续前行了。为了触发 OAuth 验证流程，处理函数唯一要做的是，在需要时调用 `RedirectToOAuth()` 方法。验证流程结束后，

我们的应用会自动把用户带到之前所在的页面。

5.2.5 处理状态的函数

走完上述 OAuth 认证流程之后，我们得到了令牌，获得了创建提交状态的权限。下面把下述处理函数添加到那个 Nancy 模块构造方法中：

```
Get["/{user}/{repo}/{sha}/{status}", true] = async (parms, ct) => ❶
    {
        var accessToken = Session["accessToken"] as string;
        if (string.IsNullOrEmpty(accessToken))
            return RedirectToOAuth(); ❷
        client.Credentials = new Credentials(accessToken);

        CommitState newState = Enum.Parse(typeof(CommitState), ❸
                                          parms.status,
                                          true);
        try
        {
            var newStatus = new NewCommitStatus ❹
            {
                State = newState,
                Context = "example-api-app",
                TargetUrl = new Uri(Request.Url.SiteBase),
            };
            await client.Repository.CommitStatus.Create(parms.user, ❺
                                                        parms.repo,
                                                        parms.sha,
                                                        newStatus);
        }
        catch (NotFoundException) ❻
        {
            return HttpStatusCode.NotFound;
        }

        var template = @"Done! Go to <a href=""https://" ❼
        + @"api.github.com/repos/{0}/{1}/commits/{2}/status"
        + @""">this API endpiont</a>";
        return String.Format(template,
                             parms.user, parms.repo, parms.sha);
    };
```

❶ 注意这个处理函数的请求路径：向 `localhost:8080/user/repo/<sha>/<status>` 发起 GET 请求，新建状态。这样便于在浏览器中测试，不过 Web 爬虫也能在不知不觉中轻易触发这个 API。对这个示例来说，可以这么做，不过在真正的应用中或许应该使用 POST 请求。

❷ 这里用到了 OAuth 辅助方法。如果会话 cookie 中没有授权令牌，重新走一遍 OAuth 验证流程。如果 Octokit 请求 API 时出现授权异常，那么也要走一遍验证流程，不过这里没有给出相关的代码。

❸ 这里尝试解析请求 URL 中的最后一段，将其保存为 `CommitState` 枚举的一个成员。Octokit 会努力为所有 API 维护类型安全，因此不能直接使用原始字符串。

❹ `NewCommitStatus` 对象封装新建状态时能设置的所有数据。这里我们设置前面解析得到的状态、用于标识服务的唯一（但愿如此）上下文值，以及不是很有用的目标 URL（实际上应该是解释如何得出结果的地址）。

❺ 这是新建状态的 REST 调用。这是个异步方法（`async`），因此在处理结果之前要等待（`await`）结果。

❻ 这个 API 可能抛出很多异常，不过我们要处理的最严重的一个是 `NotFoundException`，对应于 HTTP 404 状态码。这里，我们把代码转换回来，以提升用户体验。

❼ 如果请求成功，那么渲染一段 HTML，作为处理函数的返回值。Nancy 默认把响应的 `content-type` 设为 `text/html`，因此用户会看到可以点击的链接。

到此结束！如果你跟着我一起做，在自己的项目中输入了这些代码，那么现在可以在调试器中运行或者在 ASP.NET 服务器中托管这个应用，然后在浏览器中打开 URL，为你项目中的提交创建状态。

前面说过，不过还是要重申：这个示例使用 GET 请求是为了便于测试，真正的服务应该使用 POST 请求。

5.3　小结

即便没写多少代码，读完这一章也还是能学到不少概念。

本章中我们学习了 Commit Status API，见识了如何通过持续集成软件使用该 API。不过我们知道，除此之外 Commit Status API 还有很多用途。我们了解到，状态可以读写，API 状态可以把多个状态合并成一个通过或失败状态；如果默认的状态合并方式不符合需求，那么我们还可以自己编写规则。我们还了解了拉取请求中绿色对号和红色错号背后的机制。

本章介绍了 OAuth Web 流程，以及这么设计的原因。GitHub API 的很多功能都依赖 OAuth，这是授信和授权的正确方式。有了 OAuth，我们就可以写出与 GitHub 交互的一流应用，既能运行在 Web 中，也能运行在用户的设备中。

本章涵盖了一些 C# 知识，包括包系统、至少一个 IDE、匿名函数、对象初始化程序，等等。C# 是一门相当不错的语言，用过一段时间之后再用其他语言或许会怀念它的某些特性。

本章介绍了 .NET 包管理器 NuGet 的用法，以及这个生态系统中的几个包。这个功能异常强大，很多常见的需求都有对应的库，此外还有诸多实现不常见需求的库，因此不管你想做什么，基本上都能找到可以帮助你的 NuGet 包。

本章还介绍了 Nancy。使用这个库可以通过简洁的句法和直观的对象模型快速构建任何 HTTP 服务，从 REST API 到基于 HTML 的界面不一而足。如果你从未站在 Sinatra 的角度上观察世界，那么 Nancy 可能会稍微改变你对 Web 服务器的看法；如果你曾这么做过，那么读完本章后你将对如何使用符合习惯的方式实现这种模型有新的认识。

最后，本章介绍了 Octokit，这个库是对 REST API 的封装，类型安全，而且内建异步支持和 OAuth 辅助功能。有了这个库，处理 GitHub API 特别简单直观，就像使用任何 .NET 库一样，甚至可以通过直觉推知用法。

现在该重回 Ruby 的怀抱了。下一章介绍 Jekyll（GitHub Pages 使用的就是它），以及如何使用它搭建博客。

第6章

Ruby和Jekyll

Jekyll 项目的描述说它是一个“使用 Ruby 开发的静态博客网站生成器”。Jekyll 的核心是一系列用于构建网站的简单技术。简单是 Jekyll 的一大特点，使用 Jekyll，我们无需学习数据库后端技术，不用在服务器中安装复杂的软件，也不用像大多数构建网站的技术那样执行各种操作。很多著名的技术博客使用的博客引擎都是 Jekyll。

与 GitHub 大量使用的许多开源技术一样，Jekyll 最初是由 GitHub 的联合创始人 Tom Preson Warner 开发的，此外还有 37 Signals 的 Nick Quaranto，不过现在 Jekyll 项目有几千个贡献者。不出所料，Jekyll 的优势不在于最初的开发者有多么才智过人，也不在于想法有多么出类拔萃，而在于最初的开发者营造了和谐的社区氛围，吸引用户参与其中。

6.1　学习使用Jekyll构建博客

本章探讨 Jekyll 博客的结构，说明涉及的几个主要技术。熟悉 Jekyll 之后，我们将使用命令行工具从头开始创建一个 Jekyll 博客。然后，我们将编写一个 Ruby 程序，爬取博客类网站，把爬到的信息转换成一个新的 Jekyll 博客。

6.2　Jekyll是什么

Jekyll 把满足特定结构的文件转换成 HTML，它建立在两个可靠的工具之上。其一，Markdown，这是一种标记语言，阅读性和表现力极强。其二，Liquid Markup，这是一门简单的编程语言，为构建现代化的网页提供了足够的组件，如条件判断和循环，而且足够安全，能在公共服务器中运行任何页面。有了这两个技术的支持，加上对布局结构的约

定，无需复杂的文件结构和技术，Jekyll 就能构建特别复杂的网站。

GitHub 把 Jekyll 博客存储在 Git 仓库中，因此原生支持 Jekyll。如果 GitHub 把仓库视作 Jekyll 网站，推送文件后，GitHub 会自动重新构建网站。Jekyll 是一个开源生成器，为源文件定义了特定的结构，这个结构对其他工具而言也易于理解和处理。这意味着，我们可以自行构建工具，与 Jekyll 博客交互。Jekyll 这样的开源工具加上 GitHub API 这样优秀的 API，得到的是强大的发布工具。

在本地操作Jekyll

若想使用 Jekyll，需要安装 `jekyll` 这个 gem。如附录 B 所述，安装 Ruby gem 的方法是执行下述命令：

```
$ gem install jekyll
```

这样安装有两个问题。首先，在命令行中运行命令之后，命令消失了（不过私人的 shell 历史文件中有）。其次，如果要把网站发布到 GitHub 中，我们得确保 Jekyll 及其依赖的版本完全匹配，并确保在本地笔记本中正常的网站发布到 GitHub 中也正常运行。如果不考虑这一点，那么你时不时会收到来自 GitHub 的电子邮件，如下所示：

```
The page build failed with the following error:

page build failed

For information on troubleshooting Jekyll see
https://help.github.com/articles/using-jekyll-with-pages#troubleshooting
If you have any questions please contact GitHub Support.
```

这两个问题的解决方法相同。你可能见过其他章节使用 `Gemfile` 文件安装 Ruby 库。这种方式不在命令行中使用 `gem` 命令手动安装依赖，而是把依赖放入 `Gemfile` 文件中。这样，使用仓库的其他人只需执行 `bundle install` 命令就能安装正确的依赖。此外，我们不直接使用 `jekyll` gem，而是使用 `github-pages` gem。这个 gem 使用的 Jekyll 版本与 GitHub 使用的一致。如果你收到了上述电子邮件，执行 `bundle update` 命令，确保安装的依赖正确且同步，一般这又会在本地产生问题，但这时解决这些问题要快很多，执行如下代码即可。

```
$ printf "gem 'github-pages' >> Gemfile
$ bundle install
```

使用 Gemfile 文件创建和管理依赖是明智的做法，这样能确保本地使用的 Jekyll 与 GitHub 中运行的版本一致。

现在可以创建 Jekyll 博客了。

6.3 使用Jekyll快速创建博客

安装好所需的工具之后，我们来创建一个简单的博客。执行下述命令：

```
$ jekyll new myblog
$ cd myblog
```

`jekyll new` 命令用于创建微型 Jekyll 博客所需的基本结构。看一下那个目录，你会见到组成 Jekyll 博客基本结构的几个文件。

`jekyll new` 命令创建了两个 CSS 文件，一个用于装饰博客（main.css），另一个用于高亮句法（syntax.css）。记住，网站完全在你的掌控之下。main.css 文件只是样板，如果不符合你的需求，完全可以将其删除。如果博客中有代码片段，syntax.css 文件能高亮句法，美化多数编程语言。

新建博客时还会生成一个 .gitignore 文件，里面只有一行内容：_site。使用 Jekyll 在本地构建网站时，默认情况下所有文件都会构建到 _site 目录里。.gitignore 文件的作用是避免把那些文件纳入仓库，因为把文件推送到 GitHub 中之后，构建网站的 Jekyll 命令会覆盖那些文件。

> `jekyll new` 命令不会创建并初始化 Git 仓库，如果想要这么做，那就要执行 `git init` 命令。这个命令会创建合适的结构，便于把所有文件都添加到 Git 仓库中。我们只需执行 `git add .; git commit` 命令，.gitignore 文件会被纳入仓库，并且配置仓库忽略不需要的文件，例如 _site 目录。

所有博客文章都存储在 _posts 目录里。不是所有 Jekyll 网站都有 _posts 目录（Jekyll 可用于创建任何类型的静态网站），但是如果这个目录中有文件，Jekyll 会使用特殊的方式处理那些文件。现在打开 _posts 目录，你会看到 Jekyll 初始化命令创建的第一篇文章，文件名类似于 _posts/2014-03-03-welcome-to-jekyll.markdown。文章的名称有特定的格式：先是日期，然后是文章标题（用连字符取代空格）和扩展名（对 Markdown 文件来说是 .markdown 或 .md，对 Textile 文件来说是 .textile）。

此外，新建的 Jekyll 博客还有几个 HTML 文件：index.html 文件，即博客的首页；几个布局文件，用于套入生成的内容。进入 _layouts 目录，注意里面有个名为 default.html 的文件和名为 post.html 的文件。这两个是布局文件，用于套入生成的全部内容，例如使用 Markdown 格式编写的博客文章。比如说，post.html 文件用于套入 _posts 目录中各个文件生成的内容。首先，把标记语言编写的内容转换成 HTML，然后套入布局。看一下 _layouts 目录中的各个文件，你会发现每个文件中都有 `{{ content }}` 这样的占位符。这个占位符会被替换成其他文件生成的内容。

其实，这些占位符本身也是一种标记语言，叫 Liquid Markup，由 Shopify.com 开发和开源。Liquid Markup 之所以出现，是因为人们需要一种在模板中嵌入编程结构（如循环和变量）的安全方式，而且不能向成熟的编程环境开放渲染上下文。Shopify 需要一种允许不可信用户在面向公众的系统中上传动态内容的方式，而且不用担心标记语言放任恶意操作。例如，如果使用功能全面的嵌入式编程语言，Shopify 会面临被攻击的危险，因为用户可以编写代码连接内网中的网站。PHP 和 ERB（嵌入式 Ruby 模板，Ruby on Rails 框架的用户喜欢使用）等模板语言支持嵌入各种代码片段，这样虽然提供了特别强大的功能，能完全控制源码文档，但是用户可能会嵌入 `system("rm -rf /")` 这样的代码，执行危险操作。Liquid Markup 提供了嵌入式编程模板的诸多优点，同时规避了危险。本章后面会举几个例子，说明如何使用 Liquid Markup。

最后，Jekyll 网站的目录中有个名为 _config.yml 的特殊文件。这是 Jekyll 的配置文件。打开它，你会看到几个特别简单的设置：

```
name: Your New Jekyll Site
markdown: redcarpet
highlighter: pygments
```

这个文件中只有三行内容，各行的作用易于理解，分别设定网站的名称、Jekyll 命令使用的 Markdown 解析程序和是否使用 `pygments` 高亮句法。

若想在本地查看网站，执行下述命令：

```
$ jekyll serve
```

这个命令会构建整个 Jekyll 网站目录，然后启动一个微型 Web 服务器，伺服网站。在 Web 浏览器中访问 http://localhost:4000，你会看到网站的首页中显示了一些内容，还列出了一篇博客文章，如图 6-1 所示。



图 6-1：最简单的 Jekyll 网站

点击“Blog Posts”部分里的链接，你会看到网站中的第一篇文章，如图 6-2 所示。



图 6-2：一篇示例文章

这篇文章是由 Jekyll 初始化命令创建的。这篇文章背后的 Markdown 文件在前面介绍的 _posts 目录里：

```
---
layout: post
title:  "Welcome to Jekyll!"
date:   2014-03-03 12:56:40
categories: jekyll update
---
```

这篇文章在 _posts 目录里，你可以编辑然后重新构建（或者执行 `serve` 命令时加上 `-w` 开关），查看改动的内容。如果想新建文章，那么只需在 _posts 目录中添加文件，并且遵守这个命名约定：YYYY-MM-DD-name-of-post.ext。

Jekyll 还为代码片段提供了强大的支持：

```
{% highlight ruby %}
def print_hi(name)
  puts "Hi, #{name}"
end
print_hi('Tom')
#=> prints 'Hi, Tom' to STDOUT.
{% endhighlight %}
```

Jekyll 的更多功能说明参见文档（http://jekyllrb.com/）。如果发现缺陷，或者需要什么功能，那么在 GitHub 中 Jekyll 的仓库里创建一个工单（https://github.com/mojombo/jekyll）。

希望你也同意，与编写原始的 HTML 相比，这样创建网站更直观、更易于理解。简单和易于理解是 Jekyll 的主要优势之一。源文件易于理解了，我们就能专注于内容本身，而不用分心去管美化内容的技术。下面分析这个文件，探讨几个重要的部分。

6.3.1 YAML格式的头部元信息

Jekyll 文件的开头是 YAML 格式的头部元信息（YAML Front Matter，YFM）：

```
---
layout: post
title:  "Welcome to Jekyll!"
date:   2014-03-03 12:56:40
categories: jekyll update
---
```

YFM 是一段 YAML（YAML Ain't Markup Language），前后都有三个连字符。YAML 是一种简单的结构数据序列化语言，许多开源项目用它代替 XML，因为很多人觉得它比 XML 易于阅读和编辑。这个文件中的 YFM 展示了几个配置选项：布局（`layout`）、标题（`title`）、日期（`date`）和分类列表（`categories`）。

布局指代 _layouts 目录中的某个文件。如果 YFM 中没有指定布局文件，那么 Jekyll 假设你想使用名为 default.html 的文件套入内容。可以看出，我们可以在那个目录中添加布局文件，然后在 YFM 中覆盖默认值。这个文件指定的布局是 `post`。

标题用于生成 `<title>` 标签，此外还可以使用 Liquid Markup 提供的双花括号句法 `{{ page.title }}`，在模板中的任何位置插入标题。注意，_config.yml 文件中的变量使用 `site.` 命名空间引用，而 YFM 中的变量使用 `page.` 命名空间引用。这个文件中的标题虽然与文件名一致（空格替换成了连字符），但是修改 YFM 中的标题不会影响 Jekyll 生成的 URL。如果想修改 URL，那就要重命名文件本身。这个特性不错，因为我们可以稍微修改标题，而不影响当前的 URL。

这个 YFM 中还有两个变量：日期和分类。这两个变量完全可选，奇怪的是，Jekyll 初始化命令生成的结构和模板默认没用到它们。这两个变量为文章提供了额外的信息，不过只存储在 Markdown 文件中，生成的内容中则没有体现。分类列表经常用于生成一个索引文件，在里面列出各个分类名下的文章。如果你以前用过 WordPress，那么你有可能接触过分类。在 WordPress 中，请求分类列表时会从 MySQL 数据库读取数据，动态生成分类索引；而在 Jekyll 中，分类索引文件是静态生成的。如果想动态生成，可以创建一个 JSON 文件，存储分类和名下的文件，然后构建一个 JavaScript 小组件，在客户端请求那个文件，与之交互。通过模板，Jekyll 能生成 JSON 文件（或其他任何格式），而不仅仅局限于生成 HTML 文件。

YFM 完全是可选的。没有 YFM 的文章或页面，Jekyll 也会将其渲染到网站中。如果没有

YFM，那么页面会使用变量的默认值渲染，因此为了防止没指定布局，至少要确保默认的布局能套入所有页面。

YFM 有个重要的默认变量——published（是否发布）。这个变量的默认值是 true。也就是说，在 Jekyll 网站仓库中创建一个文件后，如果没有手动设置 published，那么该文件默认会发布。如果设为 false，那就不会发布。在私有仓库中，我们可以把这个变量设为 false，从而保证文章在完全写完之前对外不可见。可惜，协助 Jekyll 创建 Markdown 文件的工具都不会在 YFM 中显式设置 published 变量，因此，如果有不想发布的文章，在提交文件之前记得要确认一下。

6.3.2　Jekyll使用的标记

YFM 下面是 Markdown 文件的结构。最简单的 Markdown 文件只包含文本信息，没有任何格式化符号。其实，如果布局得当，即使没有任何花哨的格式，只用纯文本内容也能写出优秀的博客文章。

不过，使用少量的 Markdown 句法也可以为文章增光添彩。我们注意到的第一个 Markdown 句法是反引号，这个句法用于包围短小的代码（或者类似代码的信息，如文件名）。使用得多了，你便会发现 Markdown 蕴含的智慧，Markdown 提供的格式化符号没有 HTML 标签那么繁琐，却能实现同样的格式。

链接可以使用 [format][link] 格式，其中 link 是完全限定的 URL（如 http://example.com），或者是对页面底部某个链接的引用。这个页面中有两个引用，分别是 jekyll-gh 和 jekyll；我们可以在页面中使用 [Jekyll's GitHub repo][jekyll-gh] 这样的句法引用那两个链接。使用引用有个额外好处，即可以使用简称多次引用同一个链接。

Markdown 为各级标题提供了简单的句法，不过那篇示例文章中没有演示。添加标题的方法是使用 # 符号，如果想使用次级标题，重复输入 # 符号即可。这些定界符对应于 h 标签，两个 # 符号对应 <h2> 标签。若想把文本使用 <h3> 标签围起来，要使用 ### Some Text 这样的句法。如果觉得在行尾写上相同数量的井号（### Some Text ###）更具表现力，可以这么做，但这不是强制要求。

Markdown 为大多数 HTML 元素提供了简洁的句法，包括有序列表、无序列表、强调，等等。如果找不到相应的 Markdown 句法，还可以直接在 Markdown 格式化符号旁嵌入常规的 HTML。使用 Markdown 写东西时最好在手旁放一份 Markdown 速查表（https://github.com/adam-p/Markdown-here/wiki/Markdown-Cheatsheet）。Markdown 由 Daring Fireball 公司的 John Gruber 发明，他的网站（http://daringfireball.net/）对 Markdown 的用法和缘由做了更为深入的说明。

6.3.3　使用Jekyll命令

执行 `jekyll --help` 命令会显示 Jekyll 命令的选项。我们已经用过 `jekyll serve` 命令，它的作用是构建文件，写入 _site 目录，然后以此为根目录启动 Web 服务器。如果使用这个命令构建 Jekyll 网站，你可能还想学习几个开关。

如果经常编写和调整页面，然后切换到浏览器查看结果，你会发现 `-w` 开关（“watch”，监视）很实用。指定这个开关后，只要改动了源文件，整个网站就会自动重新构建。编辑并保存一个文章文件之后，那个文件会自动重新构建。不指定 `-w` 开关的话，必须先关闭 Jekyll 服务器，然后重新启动。

> Jekyll 监视命令可以重新加载所有的 HTML 和标记文件，但是不能重新加载 _config.yml 文件。如果修改了该文件，需要关闭再重启服务器。

如果在同一台笔记本电脑中运行多个 Jekyll 网站，第二次执行 `jekyll serve` 命令会失败，因为后一个服务器不能打开 4000 端口。此时，可以执行 `jekyll serve --port 4010` 命令，打开 4010 端口（或者任何其他想使用的端口）。

6.3.4　在Jekyll中设定隐私级别

GitHub 中的 Jekyll 网站仓库可以是公开的，也可以是私有的。在公开的仓库中，可以托管由 Jekyll 源文件生成的公开内容，而不直接托管源文件本身。记住，如前所述，YFM 中没有 `publishing: false` 的文件一旦推动到仓库中就公开了。

6.3.5　主题

Jekyll 原生不支持主题，不过可以轻易添加 CSS 文件或整个 CSS 框架。此外，如果你喜欢某个 Jekyll 博客的主题，还可以派生那个仓库。本章后面会说明如何添加自定义的 CSS，以及将其添加到何处。

6.3.6　发布到GitHub中

创建好博客之后，可以轻易将其发布到 GitHub 中。Jekyll 博客有两种发布方式：

- 作为 github.io 的子站
- 放在自己的域名名下

GitHub 为个人博客提供了托管功能，放在 github.io 域名名下。此外，稍微配置之后还能使用自己的域名托管网站。

在GitHub.io名下托管Jekyll博客

若想在 github.io 名下托管个人博客网站，Jekyll 博客应该放在 Git 仓库的 master 分支中，而且 GitHub 仓库应该命名为 `username.github.io`。一切设置得当之后，为仓库添加 GitHub 远程地址，然后推送文件即可。如果使用 `hub` 工具（与 Git 和 GitHub 交互的命令），整个过程只需几个简单的命令。记得要根据自己的用户名修改第一行。

> 起初，hub 工具使用 Ruby 编写，因此安装方法很简单，执行 `gem install hub` 命令即可，不过后来用 Go 语言重写了。Go 语言的安装过程复杂些，这里不做说明。如果你的 OS X 系统中有 `brew` 命令，可以执行 `brew install hub` 命令安装 hub。在其他平台中的安装方法各异，请参照 http://github.com/github/hub，确定你所用系统的最佳安装方式。

把 Jekyll 博客托管到 github.io 名下的方法是执行下述几个命令。

```
$ export USERNAME=xrd
$ jekyll new $USERNAME.github.io
$ cd $USERNAME.github.io
$ git init
$ git commit -m "Initial checkin" -a
$ hub create  # 这里需要登录……
$ sleep $((10*60)) && open $USERNAME.github.io
```

倒数第二行在 GitHub 中创建一个仓库，名称与本地目录相同。最后一行休眠 10 分钟，让 GitHub 配置你的 github.io 网站，然后在浏览器中打开网站。GitHub 首次配置网站可能需要十分钟，不过后续推送能立即看到新内容。

6.3.7 托管在自己的域名名下

如果想在自己的域名名下托管博客，要使用仓库的 `gh-pages` 分支。[1] 此外，还要在仓库中创建一个 CNAME 文件，并设置 DNS，把你的域名指向 GitHub 服务器。

1. gh-page分支

为了使用 `gh-pages` 分支，要在仓库中检出并创建这个分支。

```
$ git checkout -b gh-pages
$ rake post title="My next big blog post"
$ git add _posts
$ git commit -m "Added my next big blog post"
$ git push -u origin gh-pages
```

注 1：作者所说的有误，像 6.3.6 节那样放在 master 分支中也可以自定义域名。这一章对 Jekyll 的说明很多都过时了，如果想学习最新的技术，或者想进一步了解 Jekyll，可以阅读译者写的《Jekyll 小书》（http://www.ituring.com.cn/book/1833）。——译者注

记住，一定要使用 `gh-pages` 分支。如果仓库只用于存储博客，这一点可能不会忘记。添加 `-u` 开关的目的是让 Git 记住每次都推送到 `gh-pages` 分支。

2. CNAME文件

CNAME 文件是简单的文本文件，里面的内容是域名。

```
$ echo 'mydomain.com' > CNAME
$ git add CNAME
$ git commit -m "Added CNAME"
$ git push
```

把 CNAME 文件推送到仓库中之后，可以访问仓库的管理页面，确认 GitHub 是否正确地把你的域名和博客关联起来。为了快速打开管理页面，我们可以使用 `github` gem。这个 gem 已经没人维护了，不过仍是个有用的命令行工具。

```
$ gem install github
$ github admin # 打开https://github.com/username/repo/settings
```

`github` gem 是个有用的命令行工具，可惜针对的是旧版 GitHub API，因此文档中说明的功能常常是错的。

如果设置正确，在设置页面的中部会看到如图 6-3 所示的内容。

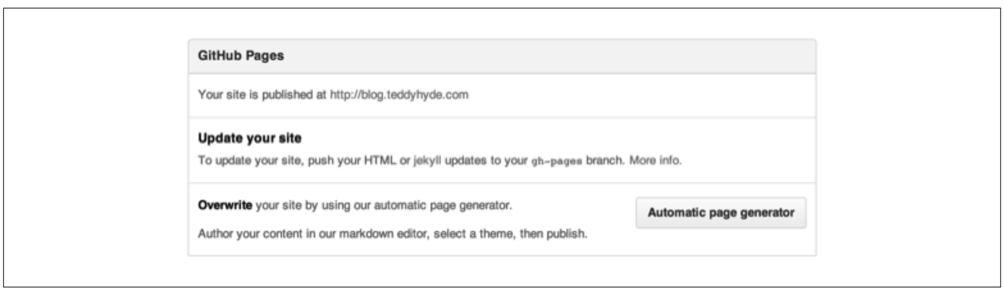


图 6-3：Jekyll 博客的设置

GitHub 正确识别了 CNAME 文件，因此会在自己的服务器中处理发往那个域名的请求。不过还没结束，我们要为网站设置 DNS。

3. 设置DNS

为网站设置 DNS 通常很简单。如果设置的是子域，而不是裸域（apex domain），那更简单。具体而言，裸域是指 mypersonaldomain.com 这样的域名，子域是指 blog.mypersonaldomain.com 这样的域名。

为博客设置子域的方法很简单：在 DNS 中创建一个 CNAME 记录，指向 username.github.io。

裸域的设置稍微复杂一点，必须创建 DNS A 记录，指向 IP 地址 `192.30.252.153` 和 `192.30.252.154`。这是 GitHub 目前使用的 IP 地址，以后可能会变。因此，使用裸域有一定风险。如果 GitHub 被迫更换 IP 地址（比如说遇到拒绝服务攻击），那么你也要改，而且要等待 DNS 传播开来。而使用子域的话，如果 GitHub 更改了 IP 地址，那么 CNAME 记录能自动指向正确的 IP 地址。[2]

6.4 导入其他博客

把现有的博客导入 Jekyll 有很多工具可用。Jekyll 博客中的文件都有约定的格式，因此我们只需获取各部分内容（文章本身和相关的元数据，如文章标题、发布日期，等等），然后将其写入文件。Jekyll 推荐使用 Markdown 编写博客内容，但是也支持 HTML 格式的内容，所以通常毫不费力就能转换博客，此外还有一些优秀的工具能自动转换。

6.4.1 导入WordPress

最常使用的是导入 WordPress 的工具。为此，需要 `jekyll-import` gem。这个 gem 与核心 Jekyll gem 分开分发，不过在 Gemfile 文件中指定 `github-pages` gem，然后执行 `bundle` 命令，也就安装了 `jekyll-import`。

1. 直接从数据库中导入

安装好 `jekyll-import` gem 之后，可以使用类似下面的命令转换 WordPress 博客：

```
$ ruby -rubygems -e 'require "jekyll-import";
    JekyllImport::Importers::WordPress.run({
      "dbname"   => "wordpress",
      "user"     => "hastie",
      "password" => "lanyon",
      "host"     => "localhost",
      "status"         => ["publish"]
    })'
```

这个命令从现有的 WordPress 中导入，前提是 Ruby 代码能访问数据库，要么能登录服务器，在服务器中执行上述命令，要么能通过网络访问 WordPress 博客的数据库（这样托管 WordPress 不好）。

注意 `status` 选项，这里我们指定自动发布导入的页面和文章。具体而言，每个文件的 YFM 中都会指定 `published: true`，即在 Jekyll 博客中发布页面或文章。如果想审阅各个页面和文章，可以把状态设为 `private`，这样导入 Jekyll 之后页面处于未发布状态。注意，如果仓库是公开的，标记为未发布的文章虽然不会出现在博客中，但是可以访问博客在 GitHub 中的仓库查看文章。

注 2：详情参见 GitHub 帮助文档（https://help.github.com/articles/using-a-custom-domain-with-github-pages/）。

除了上面列出的，还有很多选项。例如，默认情况下，WordPress-Jekyll 导入工具会从 WordPress 博客的数据库中导入分类，不过我们可以指定 `"categories" => false` 选项，禁止导入分类。

2. 从WordPress博客的XML文件中导入

另一种导入方法是，把整个数据库导出为一个 XML 文件，然后使用导入工具处理该文件：

```
ruby -rubygems -e 'require "jekyll-import";
    JekyllImport::Importers::WordpressDotCom.run({
      "source" => "wordpress.xml",
      "no_fetch_images" => false,
      "assets_folder" => "assets"
    })'
```

如果你无权维护服务器，可以使用这种方式导出文件。不过，这种方法对有权维护的网站也能用，而且可能比直接连接数据库更好。

若想导出 XML 文件，访问 WordPress 网站的导出页面。这个页面的路径通常是 /wp-admin/export.php，因此网址类似于 https://blogname.com/wp-admin/export.php（把“blogname.com”替换成你博客的域名）。

与很多免费工具一样，像这样导出也有一定的局限性。如果不是简单的 WordPress 网站，使用这个工具导入会丢失博客中存储的众多元数据，包括页面、标签、自定义字段和图像附件。

如果想保留元数据，或许应该考虑使用其他导入工具，例如 Exitwp。Exitwp 是一个 Python 工具，把 WordPress 网站转换成 Jekyll 网站的效果好得多，不过学习曲线较陡，选项也多。

6.4.2 从其他博客中导入

如果使用的博客程序不是 WordPress，可能也有对应的 Jekyll 导入工具。Jekyll 有许多导入工具，详情参见 jekyll-import 网站（http://import.jekyllrb.com/）。

例如，下述示例命令摘自 jekyll-import 网站，作用是导入 Tumblr 博客：

```
$ ruby -rubygems -e 'require "jekyll-import";
    JekyllImport::Importers::Tumblr.run({
      "url"            => "http://myblog.tumblr.com",
      "format"         => "html", ❶
      "grab_images"    => false,  ❷
      "add_highlights" => false,  ❸
      "rewrite_urls"   => false   ❹
    })'
```

导入 Tumblr 的插件有几个重要的选项。

❶ 输出 HTML 格式；如果想输出 Markdown 格式，设为 `md`。

❷ 设为 `true` 时，抓取图像。

❸ 设为 `true` 时，把代码块（缩进四个空格）放入“highlight”这个 Liquid Markup 标签中。

❹ 输出页面时把 Tumblr 博客中原来的路径重定向到 Jekyll 博客中新的路径。

从 Tumblr 中导出比从 WordPress 中导出容易得多。Tumblr 导出工具会爬取博客中的全部公开文章，然后将其转换成兼容 Jekyll 的文章格式。

我们知道如何使用 import.jekyllrb.com 网站提供的导入工具导入了，可是如果需要导入非标准的网站该怎么办呢？

6.5　爬取网站，导入Jekyll

Jekyll 提供了众多导入工具，能轻易把现有的博客导入 Jekyll 博客。但是，如果不是标准的博客，或者不是博客，也有方法迁移到 Jekyll。第一种方法是，仔细研究 GitHub 中的 Jekyll 导入工具源码（http://github.com/jekyll/jekyll-import），自己编写导入工具。如果计划把导入工具提供给别人使用，这或许是正确的做法，因为在这个过程中会扩展现有的 Jekyll 导入类，为其他贡献者建立导入标准。

另一种方法是输出 Jekyll 博客格式的文件。当然，这比阅读 Jekyll 导入工具及其代码库轻松得多。一开始，我是一名 Perl 程序员，始终坚信 Larry Wall（Perl 之父）的一句话：“我们鼓励你培养程序员的三个良好品质：懒惰、没耐心和骄傲。”那我们就顺应懒惰之心，选择第二种方法。我们将编写一些代码，爬取一个网站，然后从头新建一个 Jekyll 网站。在这个过程中，通过不断试错学习 Jekyll 博客的结构。

2000 年在巴西时，我创建了一个网站，名为 ByTravelers.com，这是我早期的一个旅行博客。有一天，我不幸丢失了数据库，以为网站的内容彻底消失了。一次偶然的机会，我在 Archive.org（互联网存档网站）中发现了 ByTravelers。那个网站列出了我几乎所有的文章。真正的数据库早已消失不见，通过 Archive.org 能恢复我的数据吗？

6.5.1　爬取策略

首先，我们要了解 Archive.org 提供的存档结构。打开 Archive.org，在页面中部的搜索框中输入“bytravelers.com”，然后点击“BROWSE HISTORY”。你会看到一个日历视图，显示着互联网存档网站爬取我的网站得到的所有页面，如图 6-4 所示。

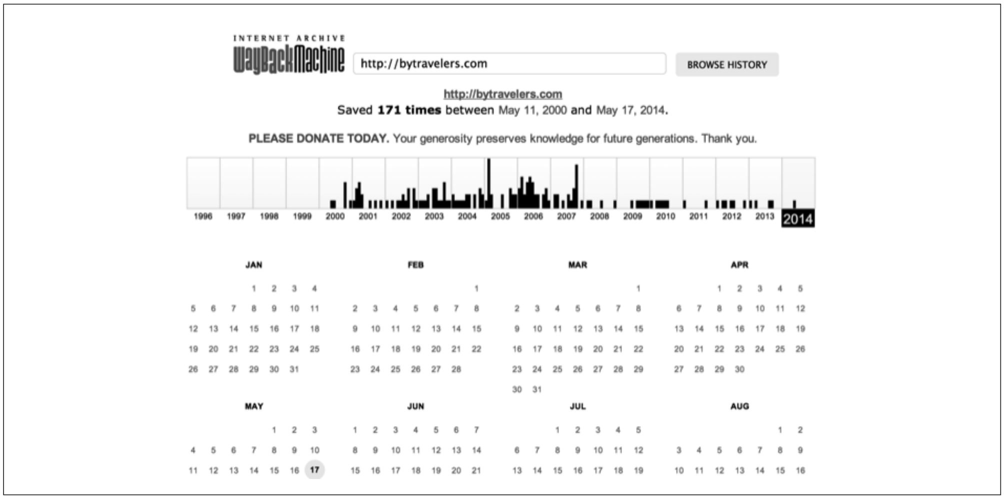


图 6-4：Archive.org 的日历视图

2003 年中，我关掉服务器，准备升级网站，换用其他技术，可是此次迁移没有完成，数据都丢失了。点击 2003 年 6 月 6 日，我们会看到那个网站的功能和数据最丰富时期的页面。虽然有几个图像无法显示，但是网站的数据被 Archive.org 网站很好地保存下来了（见图 6-5）。



图 6-5：Archive.org 中 ByTravelers.com 的存档

复制浏览器中的 URL，我们要以此作为爬取的起点。在那个页面中四处点击，我们发现各篇日志的 URL 格式统一，例如 http://www.bytravelers.com/journal/entry/56 表示网站中存储的第 56 篇日志。知道这一点之后，我们可以轻易迭代前 100 多个 URL。

6.5.2 设置

我们将实现一个简陋的爬虫，爬取过程和其他功能都写在一个 Ruby 文件中。不过，如果把功能包装成一个类，可以在另一个文件中实例化那个类，还可以编写测试，使用运行程序脚本验证爬取过程。那么，我们采用更好的做法，创建三个文件，分别用于保存爬虫类、运行程序类（实例化并运行爬虫）和测试代码（实例化并验证爬虫的功能）。

首先，编写运行程序脚本。

```
#!/usr/bin/env ruby

require './scraper'

scraper = Scraper.new()
scraper.run()
```

爬虫类的骨架如下所示。

```
class Scraper
  def run

  end
end
```

此外，还需要一个清单文件——Gemfile，列出库依赖。

```
source "https://rubygems.org"

gem "github-pages"
gem "rspec"
```

然后，执行 `bundle` 命令安装 gem，即安装 `rspec` 工具，Jekyll 工具及相关的库。

最后，创建测试用具。

```
require './scraper'

describe "#run" do
  it "should run" do
    scraper = Scraper.new
    scraper.run()
  end
end
```

记住，要执行 `bundle exec rspec scraper_spec.rb` 命令，在这个封闭的上下文中运行测试

（加载 Gemfile 中指定的库，而不是默认的系统全局 gem）。

```
$ bundle exec rspec scraper_spec.rb
.

Finished in 0.00125 seconds (files took 0.12399 seconds to load)
1 example, 0 failures
```

现在还没什么可测试，不过测试用具表明测试代码与运行程序中的代码紧密匹配。

6.5.3　爬取标题

我们从简单的事入手：从网站中爬取标题。我们将使用 Ruby 爬取网站，Ruby 有些直观的 gem，如 `mechanize`，能简化构建 Web 客户端的过程。互联网存档网站有 API，但是我发现它很奇怪而且不可靠，因此我们将直接爬取网站。执行下述命令，在 Gemfile 文件中添加一行内容，然后安装库。

```
$ echo "gem 'mechanize'" >> Gemfile
$ bundle
```

接下来，修改爬虫，让它使用 `mechanize` gem 从 Archive.org 中获取内容。

```
require 'mechanize' # ❶

class Scraper

  attr_accessor :root # ❷
  attr_accessor :agent

  def initialize # ❸
    @root = "http://web.archive.org/web/20030820233527/" +
    "http://bytravelers.com/journal/entry/" # ❹
    @agent = Mechanize.new
  end

  def run
    100.times do |i| # ❺
      url = "#{@root}#{i}" # ❻
      @agent.get( url ) do |page|
        puts "#{i} #{page.title}"
      end
    end
  end

end
```

❶ 导入 `mechanize` 库。

❷ 我们使用 Ruby 方法 `attr_accessor` 创建可公开访问的实例变量。这个方法创建的变量

可以在变量名前面加上 @ 符号访问。实例变量在类外部也可以访问。[3]

❸ 如果类中定义有 `initialize` 方法，创建对象之后会调用它。因此，这是初始化成员变量的理想之地。

❹ 初始化变量，设定默认值。我们把 ByTravelers.com 网站缓存副本的根 URL 存储在这个变量中。

❺ `run` 方法运行里面的代码块 100 次。

❻ 在这个代码块中，先生成页面的 URL，然后获取页面，最后把循环次数和页面对象的标题打印出来。

现在，运行这个爬虫，看看情况。

```
$ bundle exec ./run.rb
...
53 Read Journal Entries
54 Read Journal Entries
55 Read Journal Entries
56 Read Journal Entries
57 Internet Archive Wayback Machine
58 Internet Archive Wayback Machine
...
```

我们发现，有些条目的标题是"Internet Archive Wayback Machine"，有些是"Read Journal Entries"。Archive.org 没有存档网站的内容时会返回占位符（就像第 58 个条目那样）。我们应该忽略标题中不包括"Read Journal Entries"的页面（表明 Archive.org 已经缓存网站的内容）。

现在，我们得到了全部内容，接下来可以从中寻找重要的部分，然后将其放入 Jekyll 博客的文章中。

6.5.4　借助交互式Ruby控制台改善

Mechanize 为编写爬取工具提供了强大的基础，有两个原因：其一，便于发送 HTTP 请求；其二，为搜索远程文档提供了强大的句法。我们已经见识到如何使用 Mechanize 轻松发送 GET 请求，下面探索如何使用它过滤大量文档，获取重要的文本内容。我们可以在 Ruby IRB（交互式 Ruby shell）中手动探索爬取过程。

```
$ irb -r./scraper
2.0.0-p481 :001 > scraper = Scraper.new
 => #<Scraper:0x00000001e37ca8...>
2.0.0-p481 :002 > page = scraper.agent.get "#{scraper.root}#{56}"
 => #<Mechanize::Page {url #<URI::HTTP:0x00000001a85218...>
```

注 3：在 Ruby 中，实例变量在外部其实并不可以访问。`attr_accessor` 方法会创建实例变量，同时还会创建存取变量的方法，在类外部是通过存取方法访问变量的。——译者注

第一行启动 IRB，并使用 -r 开关加载当前目录中的爬虫库。如果你以前没用过 IRB，有几件事要知道，了解这些事之后，你的生活会变得更轻松。IRB 中有提示符，指明使用的 Ruby 版本和运行命令的序号。IRB 的功能众多，这里所涉极少。命令序号可用于重放历史，还能用于控制作业，这与很多其他类型的 shell 相似。在 IRB 提示符后面可以输入 Ruby 代码，IRB 会立即执行，然后输出结果，在 => 符号后面打印返回值。在试验 Ruby 的过程中，经常会遇到复杂的返回值，例如 scraper.agent.get 的返回值是一个 Mechanize Ruby 对象。这个对象很大，打印时要消耗大量资源。这里，我们简略了返回值的大部分内容。为了节省空间，以后讨论 IRB 时对许多复杂的对象也会这么做。

上述 IRB 中的最后一个命令把 HTTP GET 请求保存为一个 page 对象。获得这个对象之后如何从中提取信息呢？ Mechanize 提供了一个精巧的语法糖（syntactic sugar），搜索 DOM 结构特别方便，即“/”运算符。我们试试：

```
2.0.0-p481 :003 > page / "tr"
 => []
```

如果查询路径找到了内容，那么会返回一个由 Mechanize 对象组成的数组。不过，这里得到的是空数组（表明什么都没找到）。可惜，在浏览器中加载文档后，查询路径会发生变化（浏览器修改了 DOM，或者服务器为客户端发送的数据稍微不同）。但是，在 IRB 中测试类似的路径能得到所需的内容。我们可以在 Chrome 和 IRB 之间来回切换，在 Chrome 中找出 HTML 的结构后，回到 IRB 中测试搜索路径。最终，我们会得出下述搜索路径。

```
2.0.0-p481 :004 > items = page /  "table[valign=top] tr"
 => [#<Nokogiri::XML::Element:0xc05670 name="font"
      attributes=[#<Nokogiri::XML::Attr:0xc05328 name="size"
      value="-2">]...
2.0.0-p481 :005 > items.length
 => 5
2.0.0-p481 :006 > items[0].text()
 => "\n\n\n\n\n\n\n\n\n\nBeautiful Belize\n\n\n\n\n\n\n"
2.0.0-p481 :005 > items[0].text().strip
 => "Beautiful Belize"
```

就是它了，我们找到了获取标题的路径。我们要深入查询的结果，不过可以把浏览器中页面上显示的文本与 IRB 中使用查询找到的不同结构对应起来。注意，我们要去掉标题两侧的空白，这样得到的才是真正的标题。我们可以把这些代码写入爬虫，不过现在最好想想如何编写测试，确认这样做是否正确。编写测试时，还可以思考另一个问题：缓存 HTTP 请求。

6.5.5 编写测试，处理缓存

再次运行 run.rb 脚本，会发现它先打印一个文档的标题，然后停下来，从服务器中获取内容，再打印一个标题，一直重复，直到结束。Archive.org 中的内容完全不会变，因为几年

前已经爬取了源站，内容与几个月之前获取的一样，所以没有新内容要获取。因此，最好在爬虫和内容之间放一个缓存层，减少对 Archive.org 的影响，也提升爬取脚本的速度。此外，可以把获取内容和处理内容的代码分开写，便于编写测试确认两个过程。

```
require 'mechanize'
require 'vcr' # ❶
VCR.configure do |c| # ❷
  c.cassette_library_dir = 'cached'
  c.hook_into :webmock
end

class Scraper

  attr_accessor :root
  attr_accessor :agent
  attr_accessor :pages # ❸

  def initialize
    @root = "http://web.archive.org/web/20030820233527/" +
    "http://bytravelers.com/journal/entry/"
    @agent = Mechanize.new
    @pages = [] # ❹
  end

  def scrape
    100.times do |i|
      begin
        VCR.use_cassette("bt_#{i}") do # ❺
          url = "#{@root}#{i}"
          @agent.get( url ) do |page|
            if page.title.eql? "Read Journal Entries" # ❻
              pages << page
            end
          end
        end
      rescue Exception => e
        STDERR.puts "Unable to scrape this file (#{i})"
      end
    end
  end

  def process_title( row )
    row.strip # ❼
  end

  def run
    scrape()
    @pages.each do |page| # ❽
      rows = ( page / "table[valign=top] tr" )
      puts process_title( rows[0].text() )
    end
  end

end
```

❶ 导入 VCR gem。这个 gem 会拦截 HTTP 请求，正常放行首次请求，但是缓存后续请求，整个过程对用户完全透明。

❷ 使用 VCR 之前必须配置。这里，我们指定把结果缓存在哪个目录里，还告诉 VCR 使用哪个模拟库存储缓存的结果。

❸ 创建一个新变量，名为 `pages`。我们会把爬取到的所有页面存入这个数组中（缓存信息后将无偿得到它们）。

❹ 初始化 `pages` 数组。

❺ 为了使用 VCR 的录制功能，我们把发起 HTTP 请求的全部代码放入一个 VCR 块里，并为保存录制内容的 `cassette` 指定一个名称。这里，我们使用 `bt`（指代 ByTravelers）后加页面的序号来命名磁带。首次使用这个爬虫请求页面时，获取的内容会存入缓存。后续调用爬虫的 `get` 方法时，会从缓存的响应中获取内容。

❻ 然后，我们检查标题是否像是 Archive.org 存档的页面（使用标题的内容分辨），如果是，把页面存入 `pages` 数组，留待后面处理。

❼ 我们把处理标题的代码放入专门的方法中，即 `process_title` 方法。这里，去掉传入信息中的空格。

❽ 现在，在 `run` 方法中调用 `scrape` 方法爬取页面，然后迭代各个页面，搜索并处理标题。

我们得安装 VCR 和 webmock 两个库，因此要在 Gemfile 文件中添加这两个 gem。

```
$ echo "gem 'vcr'" >> Gemfile
$ echo "gem 'webmock'" >> Gemfile
$ bundle
```

执行 `bundle exec ruby ./run.rb` 命令运行脚本，会看到打印出了各个标题。

```
$ bundle exec ruby ./run.rb
Unable to scrape this file (14)
Unable to scrape this file (43)
Unable to scrape this file (47)
Unable to scrape this file (71)
Unable to scrape this file (94)
Unable to scrape this file (96)
Third day in Salvador
The Hill-Tribes of Northern Thailand
Passion Play of Oberammergau
"Angrezis in Bharat"
Cuba - the good and bad
Nemaste
Mexico/Belize/Guatemala
South Africa
...
```

我们打印出了错误（表明 Archive.org 没有存档那个 URL 对应的页面）。注意，缓存的副作用是脚本运行的速度快了。如果使用 `time` 命令分析节省的时间，可以看到如下结果。

```
$ time bundle exec ruby ./run.rb # 使用VCR之前
real    0m29.907s
user    0m2.220s
sys     0m0.170s
$ time bundle exec ruby ./run.rb # 使用VCR之后
real    0m3.750s
user    0m3.474s
sys     0m0.194s
```

可以看出，没用缓存之前消耗的时间多一个数量级。而且，缓存的响应是无偿得到的，在 IRB 会话中也能使用。

打印出来的标题看着不错，可是第四个有点不尽如人意，看样子某个用户把标题放入双引号里了。为了控制格式，最好把双引号去掉。下面就来做这一步，同时再编写测试确认处理方式是正确的。

```
require './scraper'

describe "#run" do
  before :each  do
    @scraper = Scraper.new
  end

  describe "#process_titles" do
    it "should correct titles with double quotes" do
      str = ' something " with a double quote'
      expect( @scraper.process_title( str ) ).to_not match( /"/ )
    end

    it "should strip whitespace from titles" do
      str = '\n\n something between newlines \n\n'
      expect( @scraper.process_title( str ) ).to_not match( /^\n\n/ )
    end
  end

end
```

运行测试用具，我们会发现有一个测试通过，有一个测试失败。

```
$ bundle exec rspec scraper_spec.rb
F.

Failures:

  1) #run #process_titles should correct titles with double quotes
     Failure/Error: expect( @scraper.process_title( ' something " with
     a double quote' ) ).to_not match( /"/ )
       expected "something \" with a double quote" not to match /"/
```

```
        Diff:
        @@ -1,2 +1,2 @@
        -/"/
        +"something \" with a double quote"
      # ./scraper_spec.rb:10:in `block (3 levels) in <top (required)>'

Finished in 0.01359 seconds (files took 0.83765 seconds to load)
2 examples, 1 failure

Failed examples:

rspec ./scraper_spec.rb:9 # #run #process_titles should correct titles
with double quotes
```

为了让测试通过，我们要修改 scraper.rb 文件中的一行代码，把双引号去掉。

```
...

def process_title( row )
  row.strip.gsub( /"/, '' )
end

...
```

现在两个测试都能通过了。心存戒心的程序员可能会担心那行代码。如果传入方法的值是 `nil`，那么这个方法会报错。即便我们可以保证绝对不会出现这种情况，最好也还是让方法安全一些。下面，我们要确保该方法不会报错，再编写一个测试，验证确实如此。

添加一个测试，断言 `process_title` 方法的参数为 `nil` 时不会报错。

```
...
it "should not crash if the title is nil" do
  expect{ @scraper.process_title( nil ) }.to_not raise_error()
end
...
```

执行 `rspec  scraper_spec.rb` 命令，得到如下错误。这在预料之中，因为我们还没修改代码。

```
..F..

Failures:

  1) #run #process_titles should not crash if the title is nil
     Failure/Error: expect{ @scraper.process_title( nil ) }.to_not raise_error()
       expected no Exception, got #<NoMethodError: undefined method
     `strip' for nil:NilClass> with backtrace:
         # ./scraper.rb:38:in `process_title'
         # ./scraper_spec.rb:20:in `block (4 levels) in <top (required)>'
         # ./scraper_spec.rb:20:in `block (3 levels) in <top (required)>'
     # ./scraper_spec.rb:20:in `block (3 levels) in <top (required)>'
```

```
Finished in 0.00701 seconds
5 examples, 1 failure

Failed examples:

rspec ./scraper_spec.rb:19 # #run #process_titles should not crash if the title
# is nil
```

只需简单一改就能修正这个问题：

```
...

def process_title( row )
  row.strip.gsub( /"/, '' ) if row
end
...
```

接下来该把文章写入文件了。

6.5.6　输出Jekyll文章

获得标题之后，可以生成实在的 Jekyll 文章了。简单起见，现在只为文章添加标题，稍后再添加其他内容。有了文章骨架，我们就能使用 Jekyll 命令行工具测试采用的做法是否正确。

首先，创建一个 Git 仓库来存放文件。Jekyll 工具会转换所有文件并写入 _site 目录中，因此我们要添加 .gitignore 文件，忽略这个目录：

```
$ git init
$ mkdir _posts
$ echo "_site" >> .gitignore
$ git add .gitignore
$ git commit -m "Initial checkin"
```

Jekyll 文件特别简单，开头是一些 YAML，后面是使用 Markdown 编写的文本内容。为了生成 Markdown 格式文章，我们将在爬虫中添加一个名为 `write` 的方法，从 Archive.org 获取并解析页面之后，把处理后的信息写入文件。

Jekyll 文章存储在 _posts 目录里。按约定，文件名中包含日期和标题，使用小写字母，而且只能有从 a 到 z 的英文字母和连字符，最后是扩展名（Markdown 文件通常使用 .md）。为了正确生成文件名，还要爬取日期，下面就来做这一步。

举两个实例。对 2001 年 1 月 12 日发布的“Cuba—the good and bad”一文来说，文件名是 `2001-01-12-cuba-the-good-and-bad.md`，而同一天发布的“Mexico/Belize/Guatemala”一文对应的文件名是 `2001-01-12-mexico-belize-guatemala.md`。为了确保按约定行事，我们要编写测试，如下所示。

```
describe "#get_filename" do
  it "should take 'Cuba - the good and bad' on January 12th, 2001" +
      " and get a proper filename" do
    input = 'Cuba - the good and bad'
    date = "January 12th, 2001"
    output = "2001-01-12-cuba-the-good-and-bad.md"
    expect( @scraper.get_filename( input, date ) ).to eq( output )
  end

  it "should `Mexico/Belize/Guatemala` and get a proper filename" do
    input = "Mexico/Belize/Guatemala"
    date = "2001-01-12"
    output = "2001-01-12-mexico-belize-guatemala.md"
    expect( @scraper.get_filename( input, date ) ).to eq( output )
  end
end
```

下面定义 `get_filename` 方法。这个方法使用 Ruby 中便利的 `DateTime.parse` 方法把日期的字符串表示形式转换成日期对象，然后使用 `strfmtime` 方法格式化日期对象，得到文件名中所需的格式。

```
...
def get_filename( title, date )
  processed_date = DateTime.parse( date )
  processed_title = title.downcase.gsub( /[^a-z]+/, '-' )
  "#{processed_date.strftime('%Y-%m-%d')}-#{processed_title}.md"
end
...
```

现在运行测试，会看到两个都能通过。

接下来，在爬虫中添加下述代码，输出文章。

```
def render( processed ) # ❶
  processed['layout'] = 'post'
  rendered = "#{processed.to_yaml}---\n\n" # ❷
  rendered
end

def write( rendered, processed ) # ❸
  Dir.mkdir( "_posts" ) unless File.exists?( "_posts" )
  filename = get_filename( processed['title'], processed['creation_date'] )
  File.open( "_posts/#{filename}", "w+" ) do |f|
    f.write rendered
  end
end

def process_creation_date( date )
  tuple = date.split( /last updated on:/ ) # ❹
  rv = tuple[1].strip if tuple and tuple.length > 1
  rv
end
```

```
  def run
    scrape()
    @pages.each do |page| # ❺
      rows = ( page / "table[valign=top] tr" )
      processed = {}
      processed['title'] = process_title( rows[0].text() )
      processed['creation_date'] = process_creation_date( rows[3].text() ) # ❻
      rendered = render( processed )
      write( rendered, processed )
    end
  end
```

❶ 定义 render 方法，把输入的处理后信息（一个散列）渲染成正确的格式，即 YAML 格式的头部元信息（YFM）和正文（现在没有）。返回值是渲染后的字符串。

❷ 在散列上调用 to_yaml 方法。这个方法由 yaml 库提供，导入的方法是 require 'yaml'（这里没有给出，不过能轻易添加到 scraper.rb 文件中；GitHub 中的示例有）。

❸ write 方法把渲染后的内容写入硬盘。这个方法首先检查有没有 _posts 目录，如果没有，那就创建一个。然后，调用 get_filename 方法获取路径，并在前面加上"_posts"。

❹ process_creation_date 方法的参数是爬取得到的页面中的一部分内容，作用是在字符串"last updated on:"处把那一部分拆分开，返回结果数组中的第二个元素。

❺ run 方法在前面用过的查询路径找到的行里查找日期和标题，构建 processed 散列。

❻ 处理构建好的 processed 散列，把渲染得到的字符串写入文件系统。

执行 bundle exec ruby ./run.rb 命令，_posts 目录中会生成多篇文章。随便选择一个打开，会看到类似下面的内容：

```
---
title: Beautiful Belize
creation_date: '2003-03-23'
layout: post
---
```

可以看出，目前文章中只有 YFM，不过这样也完全是有效的 Jekyll 文章。

接下来将使用 jekyll 命令行工具查看文章，并且排查 Jekyll 仓库中出现的问题。

6.5.7 使用jekyll命令行工具

我们可以花点时间把文件纳入 Git 仓库，然后使用 jekyll 命令行工具查看网站。在本地使用这个命令行工具可以查看新增的内容，即时发现错误（不用等发布到 GitHub 之后收到邮件通知）。如果爬虫没有正确处理从 Archive.org 获取的 HTML，或者随后没有生成正确的 Markdown 内容，那么可能会出错。

```
$ git add .
$ git commit -m "Make this into a Jekyll site"
...
$ jekyll serve --watch
Configuration file: none
            Source: /home/xrdawson/bytravelers
       Destination: /home/xrdawson/bytravelers/_site
      Generating...
     Build Warning: Layout 'post' requested in _posts/2000-05-23-third-day-in...
     Build Warning: Layout 'post' requested in _posts/2000-08-28-the-hill-tri...
     ...
                    done.
 Auto-regeneration: enabled for '/home/xrdawson/bytravelers'
Configuration file: none
    Server address: http://0.0.0.0:4000/
  Server running... press ctrl-c to stop.
```

可以看出，现在有几个问题。其一，没有“post”布局；其二，没有配置文件。下面修正这两个问题。

在根目录中添加一个名为 _config.yml 的文件。

```
name: ByTravelers.com: Online travel information
markdown: redcarpet
highlighter: pygments
```

记住，jekyll 工具不会自动重新加载配置文件，所以我们要按 Ctrl-C 键，然后重启。

接着，新建 _layouts 目录，在里面新建 post.html 文件，然后写入下述内容。

```
---
layout: default
---

<h1>{{ page.title }}</h1>

{{ content }}
```

这个 post.html 布局文件特别简单，使用 Liquid Markup 标签显示文章的标题（使用模板中有权访问的 page 对象获取）和内容（即渲染文章得到的输出）。

此外，还要创建“default”布局。因此，我们要在 _layouts 目录中新建一个文件，将其命名为 default.html。

```
<html>
<head>
<title>ByTravelers.com</title>
</head>

<body>
```

```
{{ content }}

</body>
</html>
```

这个文件的内容几乎都是 HTML，只有一个 `{{ content }}` 标签。如果在 Markdown 文件的 YFM 中把布局设为 `default`，Markdown 文本转换成 HTML 之后会套入那个布局文件。可以看出，文章文件指定的布局是 `post`，这个布局用于套入文章内容；而 post.html 布局文件指定的布局是 default.html，因此所有内容都会套入 default 布局。

添加这些文件之后，Jekyll 命令行工具会注意到文件系统发生了变动，因此会重新生成文件。现在，我们生成了文章，但是还没有主索引文件。下面来添加这个文件。

6.5.8　使用Liquid Markup编写主索引文件

现在能正确生成文章了，但是博客还没有入口页面。我们可以创建一个 index.md 文件，把博客文章全部列出来。

```
---
layout: default
---

# ByTravelers.com

Crowd sourced travel information.

<br/>

<div>
{% for post in site.posts %}
<a href="{{ post.url }}"><h2> {{ post.title }} </h2></a>
{{ post.content | strip_html | truncatewords: 40 }}
<br/>
<em>Posted on {{ post.date | date_to_string }}</em>
<br/>
{% endfor %}
</div>
```

注意，这个文件中既有 Markdown（单个 `#` 转换成 `h1` 标签），也有常规的 HTML。如果找不到所需的 Markdown 句法，随时可以在 Markdown 文件中混用常规的 HTML。

输出标签使用两对花括号围住内容（如 `{{ site.title }}`），而逻辑标签使用花括号和百分号（如 `{% if site.title %}`）。你可能猜到了，输出标签用于在页面中显示可见内容，而逻辑标签执行某种逻辑操作，比如说条件判断和循环。

上述模板既用到了输出标签，又用到了逻辑标签。`{% for ... %}` 就是逻辑标签，其作用是迭代各篇文章。Jekyll 会处理整个 _posts 目录，把数据存入 `site.posts` 变量，供页面使

用。因此，我们可以使用 `for` 逻辑标签迭代 `site.posts`。如果使用了 `{% for ... %}` 标签，那么后面要以 `{% endfor %}` 标签结束。该 `for` 循环中有几个输出标签，例如输出文章 URL 的 `{{ post.url }}`。我们还用到了过滤器，这是用于处理数据的方法。其中一个过滤器是 `strip_html`，你可能猜到了，它的作用是去掉 HTML 标签，得到纯文本。如果文章内容中包含 HTML 标签，那么一定要使用这个过滤器。我们还发现，过滤器可以“串联”：先使用 `strip_html` 过滤器处理正文，然后使用 `truncatewords:40` 过滤器截取前 40 个字符。

在浏览器中打开 http://localhost:4000，会看到一个简单的索引页面，列出几篇文章的标题，如图 6-6 所示。

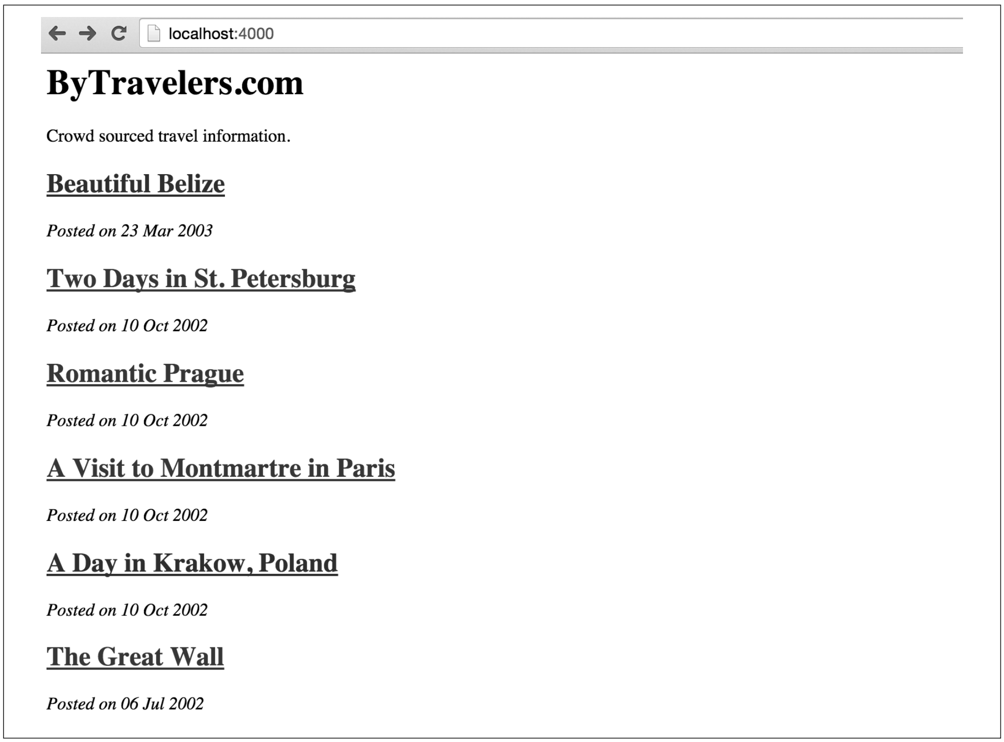


图 6-6：一个简陋的 Jekyll 博客的索引页面

这个索引页面列出了全部文章，下面我们修改一下，只列出最新的 10 篇。复制 index.md 文件，把新文件命名为 archive.md。然后，把 `{% for post in site.posts %}` 标签改为 `{% for post in site.posts | limit:10 %}`。

Jekyll 会为每篇文章生成一个页面。点击任何一个链接都会打开对应的文章，目前只有标题。接下来，我们可以让爬虫获取页面中余下的内容了。

6.5.9　爬取正文和作者

使用 IRB 查找作者和正文。首先，查找作者信息。

```
2.0.0-p481 :037 > rows[2].to_s
=> "<tr>\n<td align=\"center\">\n\n\n\n<font size=\"+1\">author:..."
2.0.0-p481 :038 > ( rows[2] / "td font" )[0].text()
=> "author: \n\nMD \n\n\nread more from this author | \nsee maps from this..."
2.0.0-p481 :039 > author = ( rows[2] / "td font" )[0].text()
=> "author: \n\nMD \n\n\nread more from this author | \nsee maps from this..."
2.0.0-p481 :040 > author =~ /author:\s+\n\n([^\s]+)\n\n/
=> 0
2.0.0-p481 :041 > $1
=> "MD"
```

我们首先查看第二行的内容，把它转换成原始的 HTML。我们发现有 `author:` 这个字符串，判断这可能是注明作者的地方。这个字符串嵌在 `font` 标签和 `td` 标签里，可以使用几个搜索路径把这些额外的信息去掉。然后，我们使用 `text()` 方法把 HTML 转换成文本，然后使用正则表达式提取 `author:` 字符串之后的文本。如果找到匹配正则表达式的内容，而且正则表达式中有捕获组，那么捕获的内容会存入全局变量 `$1` 中。当然，获取这个信息的方法不止一种。

接着，我们要从爬取的页面中获取正文。添加一个名为 `process_body` 的方法，并把获取的信息存入 `processed` 散列。

```
def render( processed )
  processed['layout'] = 'post'
  filtered = processed.reject{ |k,v| k.eql?('body') } # ❶
  rendered = "#{filtered.to_yaml}---\n\n" + # ❷
    "### Written by: #{processed['author']}\n\n" +
    processed['body']
  rendered
end
 # ❸
def process_body( paragraphs )
  paragraphs.map { |p| p.text() }.join "\n\n"
end

def run
  scrape()
  @pages.each do |page|
    rows = ( page / "table[valign=top] tr" )
    processed = {}
    processed['title'] = process_title( rows[0].text() )
    processed['creation_date'] = process_creation_date( rows[3].text() )
    processed['body'] = process_body( rows[4] / "p"  ) # ❹
    author_text = ( rows[2] / "td font" )[0].text() # ❺
    processed['author'] = $1.strip if author_text =~ /author:\s+\n\n+(.+)\n\n+/
    rendered = render( processed )
    write( rendered, processed )
```

```
    end
  end
```

❶ render 方法要稍微改一下，因为文章的 YFM 中没必要包含整个正文。可以使用 reject 方法把正文过滤掉。

❷ 然后，把作者和正文添加到渲染的输出后面。

❸ 处理正文的方式很简单：把各个段落转换成文本（使用 text() 方法），然后在各段之间插入两个换行。使用两个空行分开后，Markdown 会将其格式化成段落。

❹ 然后，只需调用 process_body 方法，并把结果写入 processed 散列。

❺ 接下来使用上述 IRB 会话中找到的那个查询路径获取作者信息，然后将其存入 processed 散列。render 方法会自动把作者名插入 YFM 中，我们再将其插入文章中。

现在可以执行 bundle exec ./run.rb 命令，重新创建文章文件。

6.5.10　把图像添加到Jekyll中

Jekyll 博客支持任何二进制文件，使用正确的标记，可以把静态资源插入 Markdown 文件中。下面把源站的图像添加到这个 Jekyll 博客中。

```
def process_image( title )
  img = ( title / "img" )
  src = img.attr('src').text()
  filename = src.split( "/" ).pop

  output = "assets/images/"
  FileUtils.mkdir_p output unless File.exists? output
  full = File.join( output, filename )

  if not File.exists? full or not File.size? full
    root = "https://web.archive.org"
    remote = root + src
    # puts "Downloading #{full} from #{remote}"
    `curl -L #{remote} -o #{full}`
  end

  filename
end
```

此处使用久负盛名的 cURL 下载图像。我们编写的代码只会在首次请求时下载图像文件。此处使用 -L 开关让 cURL 跟踪重定向，因为源站的图像显然已经在浏览器内重定向到其他 URL 了。

此处要修改 run 方法，调用 process_image 方法：在其他任何一个处理方法调用后添加 processed[‘image'] = process_image(rows[0])。

ByTravelers.com 使用的图像是我付费请画家画的。如果你使用本章的技术从其他网站爬取图像或文本内容，一定要遵守本地和国际版权法规。

然后，修改 post 布局，插入图像。

```
---
layout: default
---

<h1>{{ page.title }}</h1>

<img src="/assets/images/{{ page.image }}">

{{ content }}
```

重新生成页面，我们会看到白色背景中并列显示着有色背景的图像，这有点难看。我们可以为整个网站添加背景色，下面来修改网站的 CSS。

6.5.11　自定义样式（CSS）

这里要使用 Bootstrap，第 9 章还会再用一次。除了 Bootstrap 之外，我们还会创建一个 CSS 文件，用于定制颜色。

首先，在主布局文件 default.html 中引用 Bootstrap 和那个自定义 CSS 文件。

```
<html>
<head>
<title>ByTravelers.com</title>

<link href="/assets/css/bootstrap.min.css" rel="stylesheet">
<link href="/assets/css/site.css" rel="stylesheet">

</head>

<body>

{{ content }}

</body>
</html>
```

然后，把 Bootstrap 的 CSS 文件下载到相应的文件夹里。

```
$ mkdir assets/css
$ curl \
https://maxcdn.bootstrapcdn.com/bootstrap/3.3.5/css/bootstrap.min.css \
-o assets/css/bootstrap.min.css
```

Bootstrap 这样的 CSS 框架能提供极大的帮助，不过我们应该继续使用源站的颜色。在 assets/css 目录中新建一个文件，将其命名为 site.css。

```
body {
color: #000000;
background-color: #CCCC99;
}

a {
color: #603;
}

.jumbotron {
background-color: #FFFFCC;
}
```

安装 Bootstrap 框架之后，可以稍微修改一下 default.html 布局来美化网站。很多 Jekyll 博客的外观都十分朴实无华，可是我们不能被自己的想象所束缚。

```
<html>
  <head>
    <title>ByTravelers.com</title>
    <link href="/assets/css/bootstrap.min.css" rel="stylesheet">
    <link href="/assets/css/site.css" rel="stylesheet">
  </head>

  <body>

    <div class="container">
      <div class="jumbotron">
        <h1>ByTravelers.com</h1>
        Alternative travel information
      </div>
      <div class='row>
        <div class='span12'>
          <div class="container">
            {{ content }}
          </div>
        </div>
      </div>
    </div>
  </body>
</html>
```

重新加载页面，你会发现网站精美多了（见图 6-7）。

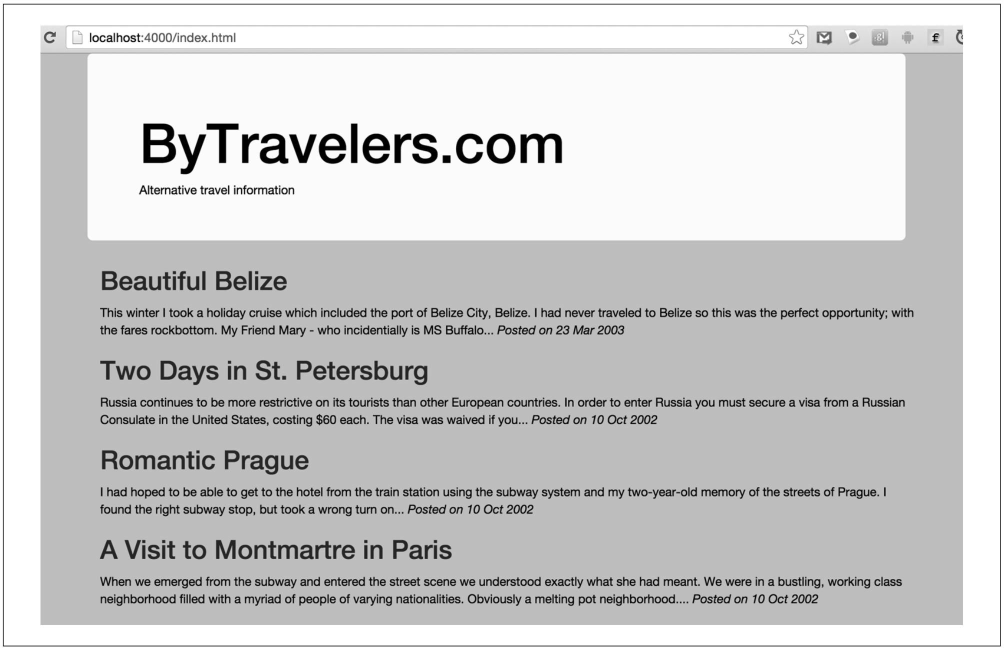


图 6-7：沿用原来的颜色和图像

至此，我们完全爬取了一个旧网站，构建了一个全新的 Jekyll 博客。于是，接下来只剩下一件事要做了——鼓励协作。有了 GitHub，这一点非常容易做到。

6.5.12　通过GitHub的“派生”功能鼓励协作

Jekyll 博客是 GitHub 中的一个仓库，这一点使得博客易于管理和跟踪变化。而且，只需点击一个按钮就能派生仓库，人们可以轻易为你的博客作贡献和修改内容。你可能见过很多托管在 GitHub 中的软件项目主页上有个“Fork me on GitHub”横幅。我们可以鼓励他人使用拉取请求参与博客的建设。最后，我们要添加那个横幅，邀请别人按照 GitHub 的方式为我们的博客做贡献。那些横幅最初在 GitHub 博客中的一篇文章（https://github.com/blog/273-github-ribbons）里发布，我们将把那里给出的代码几乎原封不动地插入 default.html 文件，唯一改动的是指向仓库的链接。

```
...
<body>

  <a href="https://github.com/xrd/bytravelers.com">
    <img style="position: absolute; top: 0; right: 0; border: 0;"
          src="https://..."
          alt="Fork me on GitHub"
          data-canonical-src="https://.../forkme_right_gray_6d6d6d.png"></a>
```

```
    <div class="container">
      <div class="jumbotron">
        <h1>ByTravelers.com</h1>
        Alternative travel information
...
```

现在，任何人都能派生我们的仓库，把他们的文章添加到 _posts 目录里，然后发起拉取请求，让我们把新文章添加到我们的 Jekyll 博客中。

6.5.13 把博客发布到GitHub中

与其他仓库一样，我们可以使用以前用过的命令把 Jekyll 博客发布到 GitHub 中。显然，你应该根据实际情况更换用户名和博客名。

```
$ export BLOG_NAME=xrd/bytravelers.com
$ gem install hub
$ hub create $BLOG_NAME # 这里可能需要登录
$ sleep $((10*60)) && open http://bytravelers.com
```

最后，别忘了设置 DNS 记录，并留出足够的时间，让记录传播开来。

6.6 小结

本章详细探讨了 Jekyll，说明了 Jekyll 博客的结构。Liquid Markup 是在 Markdown 文件中使用编程结构的强大方式，本章说明了这门模板语言的大部分重要概念。本章分析了 Jekyll 博客的内部结构，解说了复杂的 YAML 格式头部元信息（YFM），以及如何混用 HTML 和 Markdown 句法。Jekyll 博客可以使用自定义的 CSS，我们学习了如何使用完整的 CSS 框架（如 Bootstrap）搭配网站专用的小型 CSS 文件。此外，我们构建了一个爬虫应用，使用它获取了一个完整的远程网站，然后把它转换成了符合格式要求的 Jekyll 博客。虽然这个爬虫应用针对的是特定的网站，不过我们编写了测试，还使用合理的方式分离了组件，这足以证明你可以重用多数代码，快速构建自己的爬虫，并把其他网站转换成 Jekyll 博客。

下一章继续探讨 Jekyll，我们会使用 GitHub API 的 Java 绑定构建一个 Android 应用，通过 Git Data API 为 Jekyll 博客创建文章。

第7章

Android和Git Data API

目前你或许还没有把手机当成开发者工具使用，但是不久的将来你很有可能会这么做。当下的手机和平板电脑已经适合阅读代码，然而开发者在笔记本电脑中使用的编辑器还未移植到移动设备中。可是，我们正在向这个目标前进。EGit 是为 Java 开发的客户端库，编写良好，支持读取 GitHub 中存储的数据，同时也支持把数据写入 GitHub。我们可以使用这个库为当今世界上最流行的移动操作系统 Android 开发应用。

本章将使用 Java EGit 库开发一个小型 Android 应用，把文章发布到 GitHub 中托管的博客。我们使用这个博客应用登录 GitHub，简单说说当下的感受，然后应用创建 Jekyll 博客文章，再把文章推送到 GitHub 中托管的博客里。

7.1　搭建环境

为了构建这个应用，我们要创建一个 Jekyll 博客，并且安装所需的 Android 构建工具。

7.1.1　创建Jekyll博客

我们要编写添加 Jekyll 博客文章的应用，还要验证应用的行为符合预期，因此我们需要一个博客，这样才能执行相关的命令。新建 Jekyll 博客有多种方式，其中最简单的是运行本节说明的一系列 Ruby 命令。如果想进一步了解 Jekyll，请翻阅第 6 章。创建 Jekyll 博客时有几件麻烦事要注意，比如说要正确设置主机名，还要使用正确的 Git 分支。不过，这里不用考虑这些，我们只需确保仓库中有 Jekyll 博客的文件结构即可。

```
$ echo "source 'https://rubygems.org'" >> Gemfile
$ echo "gem 'github-pages'" >> Gemfile
$ echo "gem 'hub'" >> Gemfile
$ export BLOG_NAME=mytestblog
$ bundle
$ jekyll new $BLOG_NAME
$ cd $BLOG_NAME
$ hub create
$ git push -u origin master
```

上述命令先安装 Jekyll 所需的库（有一个库用于测试），然后使用 Jekyll 命令行工具生成一个新博客，最后把文件推送到 GitHub 中。在第四行，我们指定了博客的名称。这个名称可以随意修改，但是记得要相应地修改测试。

> 上述命令执行结束后应该关闭终端窗口。本章后面会在全新的目录中使用其他命令，最好别在创建 Jekyll 博客的目录中执行那些命令。我们已经把文件推送到 GitHub 中了，因此可以放心地把这个目录里的本地仓库删除。

7.1.2　Android开发工具

如果没有 Android 实体设备，那也不用惊慌。按照本章所述的方法，即使没有实体的 Android 设备，也能在虚拟设备中做开发和测试。

1. 安装Java SDK

Ruby 和 NodeJS 可以分别使用 RVM 和 NVM 通过简单的 shell 命令安装，可惜 Java 没有这样的工具。Oracle 公司控制着 Java 语言和官方 SDK 的分发，而且限制只能通过 java.oracle.com 下载。Java 可以免费获取，但是必须访问 java.oracle.com，找到符合需求的下载链接。Android 支持 Java 1.7 及以上版本。

2. 安装Android Studio

我们将使用 Android Studio，这是用于开发 Android 应用的 Google IDE。若想安装，访问 https://developer.android.com/sdk/index.html，找到针对你所用平台的下载按钮（支持 OS X、Linux 和 Windows）。Android Studio 集成了构建 Android 应用所需的全部重要工具。

7.2　新建项目

接下来，创建 Android 项目。初次打开 Android Studio 时，在右面板中会看到一个选项邀请你新建项目。点击“Start a new Android Studio project”选项，此时会打开一个界面，让你配置新项目。在“Application Name”中输入“GhRU”（“GitHub R U?”），在“Company Domain”中输入“example.com”（也可以使用你自己的域名，不过要知道，这样本章所用的目录结构就会与你的不同）。Android Studio 会自动生成“Package name”，即 `com.`

`example.ghru`。

然后，需要选择目标 SDK。选择的目标 SDK 版本越高，可供使用的 Android API 越新，不过能运行应用的设备越少。本章的代码要兼容旧版 SDK，经过权衡，我们使用 Android 4.4（KitKat），这样应用在手机和平板电脑中都能运行。根据 Android Studio 的说明，这个 SDK 目前支持世界上 49.5% 的 Android 设备，如图 7-1 所示。

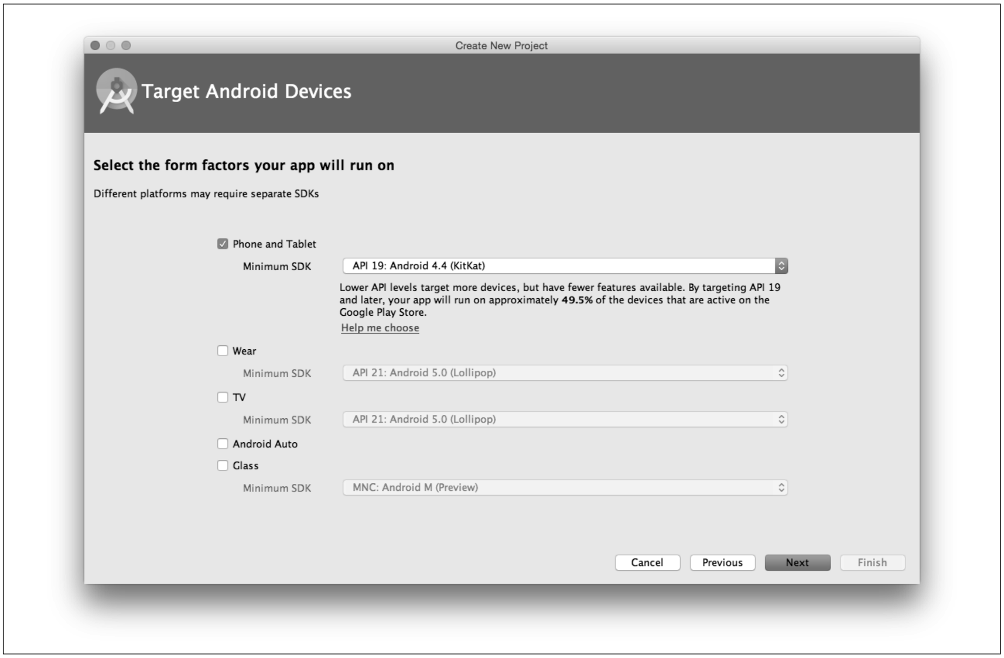


图 7-1：选择 Android SDK

接下来，Android Studio 要求我们选择活动（activity）。选择“Blank Activity”，此时会出现一个界面，让你定制活动。保留“Activity Name”（“MainActivity”）及布局、标题和菜单资源名等相关文件的默认值。最后，点击“Finish”按钮，生成项目。

设置好之后，Android Studio 会创建一个 Gradle 配置文件，并生成应用的结构。这一步完成后，我们可以点击左边纵向选项卡里的“Project”标签页，查看项目的文件树，如图 7-2 所示。

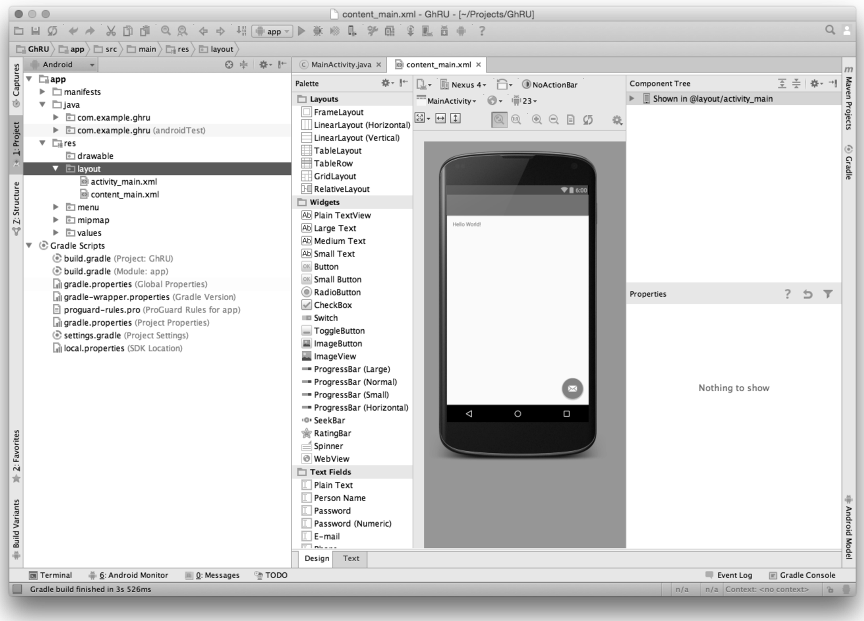


图 7-2：首次查看 Android 项目的结构

如果你从未见过 Android 项目，可能对这个界面不熟悉，需要解释一下。app 目录里是应用代码和资源（布局文件、图像和字符串）。app 目录中有几个子目录：java 目录里面显然是项目的所有 Java 代码，包括应用文件，以及把应用发布到商店中不包含在应用中，但是用于测试应用的程序；res 目录里是前面提到的资源。Android Studio 在 Gradle Scripts 部分列出所有构建文件，但是不根据目录结构分组。你会看到两个 build.gradle 文件，第一个通常可以忽略，不过我们将修改第二个。

现在，可以编辑项目了。

7.2.1　编辑Gradle构建文件

首先，我们要在 Gradle 构建文件中指定依赖库。Gradle 是针对 Java 的构建系统，现已成为 Android 平台的官方构建系统。打开 `app` 模块里的 build.gradle 文件（两个中的第二个）。

```
apply plugin: 'com.android.application' //  ❶

android {
    compileSdkVersion 23 //  ❷
    buildToolsVersion "23.0.1"

    defaultConfig {
```

```
        applicationId "com.example.ghru"
        minSdkVersion 21
        targetSdkVersion 23
        versionCode 1
        versionName "1.0"
        testInstrumentationRunner
            "android.support.test.runner.AndroidJUnitRunner" // ❸
    }
    buildTypes {
        release {
            minifyEnabled false
            proguardFiles getDefaultProguardFile('proguard-android.txt'),
            'proguard-rules.pro'
        }
    }
}

dependencies {
    compile fileTree(dir: 'libs', include: ['*.jar']) // ❹
    compile 'com.android.support:appcompat-v7:23.0.1'
    compile 'org.eclipse.mylyn.github:org.eclipse.egit.github.core:2.1.5'
    compile( 'commons-codec:commons-codec:1.9' )
    testCompile 'junit:junit:4.12' // ❺
    testCompile 'com.squareup.okhttp:okhttp:2.5.0'
    androidTestCompile 'com.android.support.test:runner:0.4' // ❻
    androidTestCompile 'com.android.support.test:rules:0.4'
    androidTestCompile 'com.android.support.test.espresso:espresso-core:2.2.1'
}
```

❶ 首先，加载 Android Gradle 插件。这个插件用于扩展项目，支持 `android` 块。接下来指定这个块。

❷ 然后，配置 `android` 块，例如目标版本（创建项目时选择的）和具体的 SDK，用于编译应用。

❸ 为了运行 UI 测试，我们要指定 `AndroidJUnitRunner` 为测试运行程序。

❹ Android Studio 自动在构建文件中添加了一个配置，加载 lib 目录里的所有 JAR 文件（Java 库）。我们还要安装兼容旧版 Android 设备的支持库，以及用于连接 GitHub 的 EGit 库（这是最重要的）。Apache 基金会开发的 Commons Codec 库提供了一些工具，用于把内容编码成 Base64，这是使用 Git Data API 把数据存入 GitHub 仓库的编码方式之一。

❺ 然后，安装只用于运行单元测试的库。使用 `testCompile` 指定的库，仅当代码在本地开发设备中运行时才编译。在这个场景下，我们需要 JUnit 库和 Square 开发的 OkHttp 库。后者用于验证提交请求确实到达 GitHub API 了。

❻ 最后，安装 Espresso 库和 Google UI 测试框架。三个库的首行安装前面配置的测试运行程序。我们使用的是 `androidTestCompile`，因此在测试模式中运行 Android 代码时才会编译这些库。

创建开发所需的AVD

使用 Android Studio 创建 AVD（Android Virtual Device，Android 虚拟设备）很简单。先打开“Tools”菜单，选择“Android”，然后选择“AVD Manager”。点击“Create Virtual Device”按钮，然后跟着向导设置。一般来说，我们可以随意选择任何设置。Google 制造了一个真实的设备，名为 Nexus 5。这是推荐使用的 Android 设备，能很好地支持所有功能。如果你不知道该选哪个，就选这个吧，如图 7-3 所示。

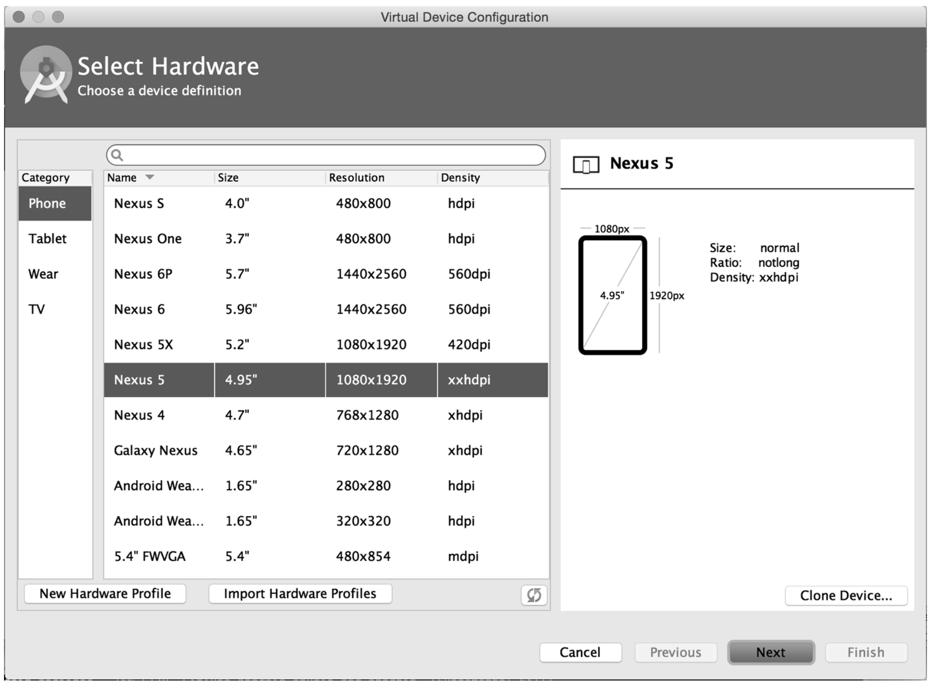


图 7-3：新建 AVD

然后，启动新建的 AVD。启动的过程要花几分钟，因为 AVD 要使用软件模拟芯片组。有别的工具能加速 AVD 的启动时间（Genymotion 是其中一个），然而，如果不使用自带的 Android 工具，复杂度会增大，因此我们依然选择使用 AVD。

7.2.2　Android默认的主活动

使用上述方式新建的 Android 应用有个示例入口，用于启动 Android 应用。所有 Android 应用都有一个名为 AndroidManifest.xml 的文件，这个文件用于指定活动，并列出应用所需的权限。打开 app/src/main 目录里的 AndroidManifest.xml 文件，添加一行代码，为这个应用赋予访问互联网的权限（因为这个应用要与 GitHub API 交互）。注意，在 Android Studio 中查看这个文件时，这个 IDE 会从资源中读取数据，篡改字符串，因此你会发现

android:label 属性的值是灰色的 GhRU，而在 XML 文件中真正显示的值如下（@string/app_name）。

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
  package="com.example.ghru">

    <uses-permission android:name="android.permission.INTERNET" />

    <application android:allowBackup="true" android:label="@string/app_name"
        android:icon="@mipmap/ic_launcher" android:supportsRtl="true"
        android:theme="@style/AppTheme">

        <activity android:name="MainActivity"
            android:label="@string/app_name">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>

    </application>

</manifest>
```

启动应用时，Android 操作系统会启动这个活动，然后调用 onCreate 函数。这个函数则调用父类的 onCreate 函数，然后具现应用的布局。布局是 XML 文件，使用声明的方式描述 Android 应用的 UI。

Android Studio 为我们创建了一个默认的布局（activity_main.xml），不过我们不用，而是新建一个。为此，在 layouts 目录上按右键（在 OS X 中按下 Control 键再点击），选择“New”，然后选择列表最顶部的“Layout resource file”（Android Studio 会根据点击的上下文呈现最优候选项）。把文件名设为 main.xml，其他设置保留默认值。

这个应用需要登录，所以我们至少需要一个用于输入用户名的字段和描述性标注，一个用于输入密码的字段（和相应的描述性标注），一个点击后尝试登录的按钮，以及一个状态字段，指明登录成功还是失败。那么，下面修改生成的 main.xml 文件，设计用户界面。为了以文本格式修改这个文件，点击 main.xml 面板底部“Design”标签页旁边的“Text”标签页，切换到文本视图。然后，把这个文件修改成下面这样。

```
<?xml version="1.0" encoding="utf-8"?> <!-- ❶ -->
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    >  <!-- ❷ -->
<TextView
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
```

```
        android:text="GitHub Username:"
        />
    <EditText
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:id="@+id/username"
        />

    <TextView
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="GitHub Password:"
        />

    <EditText
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:id="@+id/password"
        android:inputType="textWebPassword"
        />  <!-- ❸ -->

    <Button
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="Login"
        android:id="@+id/login"
        />  <!-- ❹ -->

    <TextView
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:id="@+id/login_status"
        />

</LinearLayout>
```

也许你惧怕 XML 文件（我就是这样），可是这样才便于使用声明的方式编写 Android 布局，而且有很多 GUI 工具提供了强大的 XML 布局文件管理功能。浏览一遍这个 XML 文件，我们可以较为容易地理解它的作用。

❶ 整个布局放在 `LinearLayout` 元素里。这个元素的作用很简单：纵向堆叠里面的各个元素。我们把布局的高度和宽度设为 `match_parent`，即这个布局占满整个屏幕。

❷ 然后，添加前文所述的元素：显示标注及用于输入用户名和密码的两对 `TextView` 和 `EditView`。

❸ 把密码字段的类型设为密码，遮盖里面输入的内容。

❹ 这个 XML 文件中，有些元素有 ID 属性，这样我们就能在 Java 代码中访问那些元素，例如为按钮指派处理函数，或者获取用户在字段中输入的文本。稍后会演示怎么做。

可以点击“Design”标签页，切换回设计模式，查看这个 XML 文件得到的视觉结构。

登录后也需要布局。使用相同的方式新建 logged_in.xml 文件。登录后呈现给用户的布局要让用户选择使用哪个仓库，要有一个大的文本字段用于输入博客文章，还要有一个按钮用于提交博客文章。另外还要在按钮下放一个状态框，指明文章是否成功保存。

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    >
  <TextView
      android:layout_width="match_parent"
      android:layout_height="wrap_content"
      android:text="Logged into GitHub"
      android:layout_weight="0"
      android:id="@+id/status" />

  <EditText
      android:layout_width="match_parent"
      android:layout_height="wrap_content"
      android:hint="Enter the blog repository"
      android:id="@+id/repository"
      android:layout_weight="0"
      />

    <EditText
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:hint="Enter the blog title"
        android:id="@+id/title"
        android:layout_weight="0" />

    <EditText
      android:gravity="top"
      android:layout_width="match_parent"
      android:layout_height="match_parent"
      android:hint="Enter your blog post"
      android:id="@+id/post"
      android:layout_weight="1"
      />

  <Button
      android:layout_width="match_parent"
      android:layout_height="wrap_content"
      android:layout_weight="0"
      android:id="@+id/submit"
      android:text="Send blog post"/>

</LinearLayout>
```

写过 main.xml 文件之后，你对这个文件里的大部分代码应该都熟悉了（如果不想自己输入，可以从 GitHub 中附带的示例仓库里复制）。

布局 XML 写好了，接下来可以测试应用了。

7.3　自动测试Android应用

Android 支持三种测试：单元测试、集成测试和用户界面（UI）测试。单元测试验证结构非常紧凑的独立代码片段，而集成测试和 UI 测试验证大段代码的整体行为。对 Android 来说，集成测试一般意味着要实例化数据管理器，或者让应用内的多个组件交互；而 UI 测试验证面向用户的元素，如按钮和文本字段。本章将创建一个单元测试和一个 UI 测试。

有一点要特别注意：单元测试在开发设备中运行，而不是在 Android 设备中运行。而 UI 测试在 Android 设备（或仿真器）中运行。开发设备中运行的 Java 解释器和 Android 设备中运行的 Dalvik 解释器可能有细微差别，因此三种测试都要编写。也就是说，至少要写一种在 Android 设备或仿真器中运行的测试。

7.3.1　对GitHub客户端做单元测试

我们先来编写单元测试。因为单元测试在开发设备中运行，所以测试和实现代码不能加载任何 Android 类。这一限制要求我们只能把重点放在 GitHub API 上。我们将定义一个辅助类，让它处理与 GitHub API 的交互，但是它对 Android 一无所知。然后，编写测试用具，实例化这个类，验证与 GitHub 的交互能得到正确的结果。

> 按理说，你可能会问：单元测试应该验证 API 调用吗？这种类型的测试运行速度快吗？毕竟软件开发者很快就会忽视运行速度慢的单元测试。在单元测试中模拟响应数据好吗？这些都是好问题！

为了编写单元测试，我们要把构建方式（build variant）改为单元测试。在 Android Studio 左侧的纵向选项卡中找到“Build Variants”，点击打开，然后把“Test Artifact”切换成“Unit Tests”。在项目视图（如果没有选中项目视图，点击纵向选项卡中的“Project”）中展开 java 目录，里面有个后面带有“(test)”的目录，这是存储测试的目录。如果没有这个目录，在命令行中创建（可以使用这个命令：`mkdir -p app/src/test/java/com/example/ghru`）。

然后，创建一个测试文件，将其命名为 GitHubHelperTest.java，再写入下述代码：

```
package com.example.ghru;

import com.squareup.okhttp.OkHttpClient; //  ❶
import com.squareup.okhttp.Request;
import com.squareup.okhttp.Response;

import org.junit.Test; //  ❷
```

```java
import java.util.Date;

import static org.junit.Assert.assertTrue;

/**
 * 编写单元测试之前,要在Build Variants面板中切换Test Artifact
 */
public class GitHubHelperTest { //  ❸
    @Test
    public void testClient() throws Exception {

        String login = System.getenv("GITHUB_HELPER_USERNAME"); //  ❹
        String password = System.getenv("GITHUB_HELPER_PASSWORD");
        String repoName = login + ".github.io";

        int randomNumber = (int)(Math.random() * 10000000);
        String randomString = String.valueOf( randomNumber );
        String randomAndDate = randomString + " " +
          (new Date()).toString() ; //  ❺

        GitHubHelper ghh = new GitHubHelper( login, password ); //  ❻
        ghh.SaveFile(repoName,
             "Some random title",
             "Some random body text",
             randomAndDate );

        Thread.sleep(3000); //  ❼

        String url = "https://api.github.com/repos/" +  //  ❽
        login + "/" + repoName + "/events";
        OkHttpClient ok = new OkHttpClient();
        Request request = new Request.Builder()
                .url( url )
                .build();
        Response response = ok.newCall( request ).execute();
        String body = response.body().string();

        assertTrue( "Body does not have: " + randomAndDate,   //  ❾
            body.contains( randomAndDate ) );
    }

}
```

❶ 首先，导入用于发送 HTTP 请求的 OkHttp 库。我们将查看仓库的事件日志（可以通过 HTTP 访问），验证对 GitHub API 的调用确实发送到 GitHub 了。

❷ 然后，导入 JUnit。这个库提供的 `@Test` 注解用于告知测试运行程序哪个方法是测试函数（即在测试模式中应该当作测试执行）。

❸ 定义 `GitHubHelperTest` 类。我们在这个类中只定义了一个测试用例，即 `testClient`。此处使用 `@Test` 注解告诉 JUnit，这是一个测试用例。

❹ 指定登录信息和测试用的仓库。为了不在源码中写入密码，此处使用环境变量获取运

行测试时指定的密码。

❺ 然后，构建一个随机字符串。我们把这个独一无二的字符串当作提交说明使用。这是一个信标，我们使用它验证提交一路顺畅，最终存入了 GitHub，此外还用它把这个测试创建的提交与其他测试最近创建的提交区分开。

❻ 现在，进入测试的实质阶段：传入登录凭据，实例化那个 GitHub 辅助类，然后使用 `SaveFile` 函数保存文件。最后一个参数是稍后要验证的提交说明。

❼ 有时，GitHub API 已经登记了提交，但是 API 返回的结果中却没有那次事件。睡眠几秒钟能解决这个问题。

❽ 接下来，使用 OkHttp 库发送 HTTP 请求。此处构建了一个用于获取指定仓库的事件的 URL。如果事件类型是推送，那么事件中会有提交说明。巧了，这个仓库是公开的，因此无需验证身份就能使用 GitHub API 获取数据。

❾ 检查 HTTP 响应的主体，验证里面有那个提交说明。

最后一步可能要稍做研究才能写出来。使用 cURL 请求事件 URL，会得到类似下面的数据。

```
$ curl https://api.github.com/repos/burningonup/burningonup.github.io/events
[
  {
    "id": "3244787408",
    "type": "PushEvent",
    ...
    "repo": {
      "id": 44361330,
      "name": "BurningOnUp/BurningOnUp.github.io",
      "url":
      "https://api.github.com/repos/BurningOnUp/BurningOnUp.github.io"
    },
    "payload": {
      ...
      "commits": [
        {
          "sha": "28f247973e73e3128737cab33e1000a7c281ff4b",
          "author": {
            "email": "unknown@example.com",
            "name": "Unknown"
          },
          "message": "207925 Thu Oct 15 23:06:09 PDT 2015",
          "distinct": true,
          "url":
      "https://api.github.com/repos/BurningOnUp/BurningOnUp.github.io/..."
        }
      ]
    }
...
]
```

显然，这是 JSON。可以看出，这个事件的类型是 PushEvent，而且有符合随机字符串格式的提交说明。上述 JSON 可以构建成复杂的对象结构，不过这个测试检查 JSON 字符串即可。

7.3.2 对Android应用做UI测试

接下来编写一个 UI 测试。这个测试先打开应用，找到用户名和密码字段后，输入正确的用户名和密码，然后点击登录按钮，最后检查 UI 中有没有“Logged into GitHub”文本，从而验证是否成功登录。

Android 使用 Espresso 框架支持 UI 测试。前面配置 Gradle 时已经安装了 Espresso，因此现在可以编写测试。测试要继承一个通用的测试基类（ActivityInstrumentationTestCase2），测试类中定义的所有公开函数都会作为测试运行。

在 Android Studio 的“Build Variants”面板中选择“Android Instrumentation Test”，这时会出现一个带有“androidTest”的测试目录。这个目录里的测试将在仿真器或真正的设备中运行。在这个目录中新建一个文件，将其命名为 MainActivityTest.java。

```
package com.example.ghru;

import android.support.test.InstrumentationRegistry; // ❶
import android.test.ActivityInstrumentationTestCase2;
import static android.support.test.espresso.Espresso.onView;
import static android.support.test.espresso.action.ViewActions.*;
import static android.support.test.espresso.assertion.ViewAssertions.matches;
import static android.support.test.espresso.matcher.ViewMatchers.*;

public class MainActivityTest  // ❷
    extends ActivityInstrumentationTestCase2<MainActivity> {

    public MainActivityTest() {
        super( MainActivity.class ); // ❸
    }

    public void testLogin() { // ❹
        injectInstrumentation( InstrumentationRegistry.
        getInstrumentation() ); // ❺
        MainActivity mainActivity = getActivity();
        String username = mainActivity // ❻
                .getString( R.string.github_helper_username );
        onView( withId( R.id.username ) ) // ❼
            .perform( typeText( username ) ); // ❽
        String password = mainActivity
                .getString( R.string.github_helper_password );
        onView( withId( R.id.password ) )
            .perform( typeText( password ) );
        onView( withId( R.id.login ) )
            .perform( click() );
        onView( withId( R.id.status ) ) // ❾
```

```
            .check( matches( withText( "Logged into GitHub" ) ) );

    }
}
```

❶ 导入实装登记簿（为应用的测试提供实装）、基类，以及在测试中用于下定断言的匹配器。

❷ 定义一个测试类，继承自通用的 `ActivityInstrumentationTestCase2` 类。

❸ Espresso 测试的构造方法需要调用父类的构造方法，并在参数中传入这个测试使用的活动类（这里是 `MainActivity`）。

❹ 此处测试的目的是验证可以登录 GitHub，因此起个相关的名称。

❺ 然后，加载实装登记簿，再调用 `getActivity` 方法，这样方能实例化并启动活动。如果这两步要在多个测试中使用，很多 Espresso 测试会将其写入一个使用 `@Before` 注解的函数中（以便在每个测试之前运行）。这里，为了减少函数数量，我们在唯一的测试函数中直接实现那两步。

❻ 凭据一定不能存储在代码仓库里，因此在活动上调用 `getString` 函数，从 XML 资源文件中获取用户名和密码。稍后会展示这个机密文件的内容。

❼ 获得用户名之后，将其输入界面中的文本字段里。`onView` 函数的作用是与视图（例如按钮和文本字段）交互。`withId` 函数使用 XML 布局文件中的资源标识符查找视图。找到视图后，就可以执行操作（使用 `perform` 函数），例如输入文本。这一串调用在第一个文本字段中输入 GitHub 用户名。

❽ 输入密码，然后点击登录按钮，完成与 UI 的交互。

❾ 如果成功，那么会看到“Logged into GitHub”文本。在背后，这个测试会验证我们登录了 GitHub，并显示了成功登录的结果。

为了给测试提供用户名和密码，而且不把凭据写入源码，我们要在资源文件夹中的 strings 目录里创建一个名为 secrets.xml 的文件，然后写入下述代码。

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <string name="github_helper_login">MyUsername</string>
    <string name="github_helper_password">MyPwd123</string>
</resources>
```

接下来要在 .gitignore 文件中添加一个排除规则，确保不把这个文件纳入源码仓库（快速方式是执行命令 `echo "secrets.xml" >> .gitgnore`）。

测试目前无法编译，因为我们还没编写应用的其他部分呢。因此，这里暂时不讲如何在 Android Studio 中运行测试。

下面我们要构建应用，让测试通过。

7.4 实现应用

现在可以为应用编写一些 Java 代码了。下面让 MainActivity 类具现前面定义的布局 .

```
package com.example.ghru;

import android.app.Activity;
import android.os.Bundle;
import android.widget.Button;
import android.widget.LinearLayout;
import android.widget.EditText;
import android.widget.TextView;
import android.view.View;

public class MainActivity extends Activity
{
    /** 初次创建活动时调用 */
    @Override
    public void onCreate(Bundle savedInstanceState)
    {
        super.onCreate(savedInstanceState);
        setContentView( R.layout.main);

        Button login = (Button)findViewById( R.id.login );
        login.setOnClickListener(new View.OnClickListener() { // ❶
            public void onClick(View v) {
                login(); // ❷
            }
        });
    }

    private void login() {

        setContentView(R.layout.logged_in); // ❸

        Button submit = (Button)findViewById( R.id.submit );
        submit.setOnClickListener(new View.OnClickListener() {
            public void onClick(View v) { // ❹
                doPost();
            }
        });
    }

    private void doPost() {
        TextView tv = (TextView)findViewById( R.id.post_status ); // ❺
        tv.setText( "Successful jekyll post" );
    }

}
```

这段代码模拟我们将实现的功能，并展示代码写完之后界面到底是什么样子。

❶ 为登录按钮注册点击事件的处理函数。

❷ 点击登录按钮时，调用 `login()` 函数，触发登录流程。

❸ 登录后，显示已登录布局，让用户撰写博客文章。

❹ 为提交按钮注册点击事件的处理函数。点击提交按钮时，调用 `doPost()` 函数。

❺ `doPost()` 函数更新应用底部的状态消息。

虽然代码还未实现全部功能，但是应用可以编译了。这时适合试用应用，验证 UI 的外观是否恰当。登录表单如图 7-4 所示。

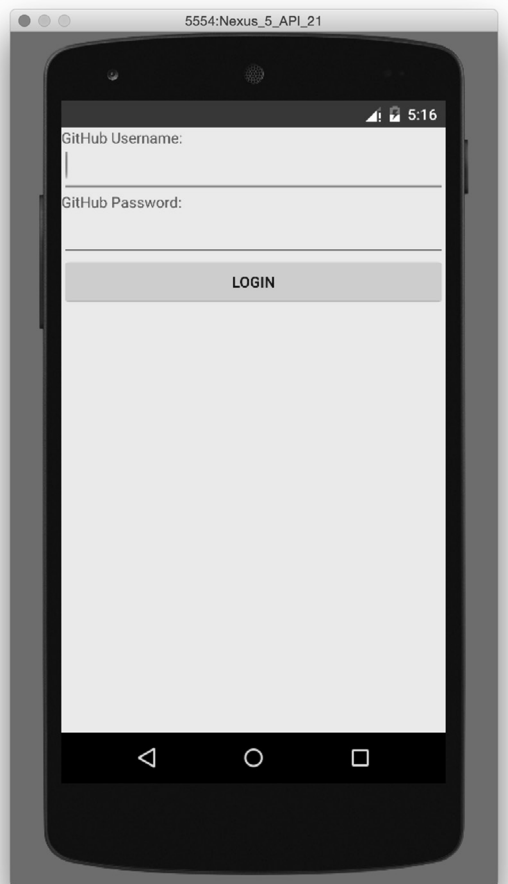


图 7-4：简单的登录 UI

7.4.1 编写登录GitHub的代码

现在可以连接 GitHub API 了。首先，实现 `login()` 函数。参照 EGit 库的参考文档（https://github.com/eclipse/egit-github/tree/master/org.eclipse.egit.github.core），登录 GitHub 的代码可以简单地写成下面这样。

```
GitHubClient client = new GitHubClient();
client.setCredentials("us3r", "passw0rd");
```

然而，运行这段代码的上下文没有代码本身这么简单。除非使用后台线程，否则 Android 操作系统不允许运行任何连接网络的代码。如果你不是 Java 开发者，可能觉得在 Java 中使用线程很麻烦，其实不然，Android SDK 提供了一个便于管理后台任务的类，名为 `AsyncTask`。这个类提供了几个入口点，我们可以方便地进入 Android 操作系统管理的线程的生命周期。我们要定义一个类，然后覆盖 `AsyncTask` 类提供的两个函数：第一个是 `doInBackground()` 函数，它把操作移到主线程之外（即后台线程中）；第二个是 `onPostExecute()` 函数，它在 UI 线程中运行，用于把 `doInBackground()` 函数的结果更新到 UI 中。

在实现登录功能之前，需要更新 `MainActivity` 类中的 `onCreate` 函数。登录按钮用于处理登录操作，因此要在登录按钮上注册一个点击事件的处理函数，调用后面即将定义的 `LoginTask`（`AsyncTask` 的子类）。

```
...
    @Override
    public void onCreate(Bundle savedInstanceState)
    {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);

        Button login = (Button)findViewById( R.id.login );
        login.setOnClickListener(new View.OnClickListener() {
                public void onClick(View v) {
                    EditText utv = (EditText)findViewById( R.id.username );
                    EditText ptv = (EditText)findViewById( R.id.password );
                    username = (String)utv.getText().toString();
                    password = (String)ptv.getText().toString(); // ❶
                    TextView status =
                     (TextView)findViewById( R.id.login_status );
                    status.setText( "Logging in, please wait..." ); // ❷
                    new LoginTask().execute( username, password );  // ❸
                }
            });
    }
...
```

❶ 从 UI 元素中获取用户名和密码。

❷ UI 应该告知用户，后台任务正在处理登录操作；因此，我们获取状态文本元素，然后

更新里面的文本。

❸ 然后，启动后台线程，处理登录操作。这一行代码会新建一个线程，传入用户名和密码作为参数。Android 会代为管理这个线程的生命周期，例如在主 UI 线程之外启动这个新线程。

下面，实现 `LoginTask` 类。

```
...
    class LoginTask extends AsyncTask<String, Void, Boolean> {   // ❶
        @Override
            protected Boolean doInBackground(String... credentials) { // ❷
            boolean rv = false;
            UserService us = new UserService();
            us.getClient().setCredentials( credentials[0], credentials[1] );
            try {
                User user = us.getUser( credentials[0] );  // ❸
                rv = null != user;
            }
            catch( IOException ioe ) {}
            return rv;
        }

        @Override
            protected void onPostExecute(Boolean result) {
            if( result ) {
                loggedIn();  // ❹
            }
            else { // ❺
                TextView status = (TextView)findViewById( R.id.login_status );
                status.setText( "Invalid login, please check credentials" );
            }
        }
    }
...
```

❶ 我们定义的这个类继承自 `AsyncTask`。泛型签名中有三个类型：`String`、`Void` 和 `Boolean`。它们分别是传给入口点的参数、中间回调和最终回调。其中，最终回调把控制器交还调用线程。第一个类型用于指定实例化任务所需的参数；我们要为后台任务提供用户名和密码，而根据签名，第一个类型可以传入字符串数组。在函数的签名中，省略号表示可以把任意多个参数（这叫变长参数）传入函数。在函数的定义体中，我们期待传入两个字符串，因此调用函数时一定要传入两个字符串。

❷ 在 `doInBackground()` 函数中，我们实例化 `UserService` 类。这个类是对 GitHub API 的包装，作用是与用户服务 API 调用交互。为了获取这个信息，需要获取服务调用的客户端，然后把用户名和密码传给客户端。这几行代码就是做这个的。

❸ 我们把调用 `getUser()` 函数的代码放入 `try` 块里，因为 `getUser()` 函数可能会抛出错误（比如说网络不可用）。其实无需使用 `User` 对象获取用户的信息，可是这么做能确保用

户名和密码是正确的，而且可以把结果存入返回值。调用 API 时 GitHub 才会使用我们设置的凭据，因此我们要访问什么东西时才能验证凭据可用。

❹ 调用的函数是 loggedIn()，而不是 login()，这样能更准确地表明此时已经登录 GitHub。

❺ 如果由于网络不可用或者凭据错误而导致登录失败，那就在状态消息中体现出来。如果用户愿意，那么可以重试。

登录成功后，loggedIn 函数会更新 UI，然后把文章发布到 GitHub 中。

```
...
    private void loggedIn() {

        setContentView(R.layout.logged_in);  // ❶

        Button submit = (Button)findViewById( R.id.submit );
        submit.setOnClickListener(new View.OnClickListener() { // ❷
            public void onClick(View v) {

                TextView status = (TextView) findViewById(R.id.login_status);
                status.setText("Logging in, please wait...");

                EditText post = (EditText) findViewById(R.id.post); // ❸
                String postContents = post.getText().toString();

                EditText repo = (EditText) findViewById(R.id.repository);
                String repoName = repo.getText().toString();

                EditText title = (EditText) findViewById(R.id.title);
                String titleText = title.getText().toString();

                doPost(repoName, titleText, postContents); // ❹
            }
        });
    }
...
```

❶ 呈现已登录布局，表明确已登录。

❷ 然后，为提交按钮注册点击事件的处理函数，这样提交文章后可以在 GitHub 中创建文章。

❸ 我们要收集用户提供的三个信息：文章正文、文章标题和仓库名。

❹ 把这三个信息传给 doPost 函数，执行异步任务。

有了使用 AsyncTask 类的经验之后，定义 doPost() 函数应该顺手了。doPost() 函数要把文章提交到 GitHub 中，还要处理后台线程中的网络活动。

```
...
    private void doPost( String repoName, String title, String post ) {
```

```
        new PostTask().execute( username, password, repoName, title, post );
    }

    class PostTask extends AsyncTask<String, Void, Boolean> {

        @Override
        protected Boolean doInBackground(String... information) { // ❶
            String login = information[0];
            String password = information[1];
            String repoName = information[2];
            String titleText = information[3];
            String postContents = information[4];

            Boolean rv = false; // ❷
            GitHubHelper ghh = new GitHubHelper(login, password); // ❸
            try {
                rv = ghh.SaveFile(repoName, titleText,
                postContents, "GhRu Update"); // ❹
            } catch (IOException ioe) { // ❺
                Log.d(ioe.getStackTrace().toString(), "GhRu");
            }
            return rv;
        }

        @Override
        protected void onPostExecute(Boolean result) {
            TextView status = (TextView) findViewById(R.id.status);
            if (result) { // ❻
                status.setText("Successful jekyll post");

                EditText post = (EditText) findViewById(R.id.post);
                post.setText("");

                EditText repo = (EditText) findViewById(R.id.repository);
                repo.setText("");

                EditText title = (EditText) findViewById(R.id.title);
                title.setText("");
            } else {
                status.setText("Post failed.");
            }
        }
    }
...
```

❶ 首先，获取需要传给 GitHub API 的参数。注意，这些信息不是从 UI 中获取的。后台线程无权访问 Android 的 UI 函数。

❷ 这个函数的返回值是 `true` 或 `false`，表明登录成功还是失败（变量 `rv` 是“return value”的缩写）。在一切未定之前，我们假定登录失败，因此先把 `rv` 的值设为 `false`。`return` 语句的值会传给线程生命周期中的下一个阶段，即名为 `onPostExecute` 的函数（这是生命周期中一个可选的阶段，我们使用这个阶段把操作状态报告给用户）。

❸ 实例化 `GitHubHelper` 类。我们应该熟悉这种实例化和使用方式了，因为单元测试中也是这么做的。

❹ 这个辅助类返回 `true` 或 `false`。如果执行到这里了，这就是最终的返回值。

❺ 为了处理错误（大多数情况下是网络错误），我们把对 `SaveFile` 函数的调用代码放入 `try/catch` 块中。

❻ `onPostExecute()` 函数是可选的，在后台任务结束后调用。这个函数的参数是前一个函数的返回值。如果 `doInBackground()` 函数的返回值是 `true`，说明文件保存成功，我们就更新应用的 UI。

我们要导入支持类。Gradle 已经自动把 EGit 的 JAR 文件和类添加到项目中了。我们要在文件的顶部，其他 `import` 语句的下面添加下述 `import` 语句。

```
...
import android.view.View;
import android.os.AsyncTask;
import org.eclipse.egit.github.core.service.UserService;
import org.eclipse.egit.github.core.User;
import java.io.IOException;
...
```

现在可以编写把数据写入 GitHub 的代码了。

7.4.2　编写与GitHub交互的代码

最后一步，编写代码把内容放到 GitHub 中。这不是件简单的事，因为 GitHub API 要求构建 Git 内部使用的结构。如果想进一步了解这个结构，推荐你阅读免费开源的《精通 Git》一书（https://progit.org/），尤其是最后一章“Git 内部原理”（http://dwz.cn/git-internals）。

简单来说，GitHub API 希望我们创建一个 Git“树”，然后在树里放一个 blob 对象。随后，把树封装到“提交”对象里，再使用数据服务封装程序在 GitHub 中创建提交。此外，若想把树写入 GitHub，需要知道基 SHA 标识符，因此你会看到用于从当前分支最后一个树中获取 SHA 的代码。不管把代码推送到 master 分支还是 `gh-pages` 分支，这部分代码都能正常运行，因此这个实用类支持 Jekyll 博客。

我们将定义一个名为 `GitHubHelper` 的辅助类，并在里面定义一个函数，用于把文件写入仓库。

按照 GitHub API 的要求，仓库里存储的文件要使用 Base64 或 UTF-8 编码。Apache 基金会在 Maven（EGit 库也是从这个软件仓库中获取的）中发布的一套工具可以执行这种编码操作，Gradle 文件已经安装了这些工具（“commons-codec”声明）。

此处先在 `SaveFile` 函数中调用几个函数，在 GitHub 中创建提交。这些函数都有一定的复

杂度，因此这里先说明如何使用 Git Data API 把数据存入 GitHub。

```
package com.example;

import android.util.Log;

import org.eclipse.egit.github.core.*;
import org.eclipse.egit.github.core.client.GitHubClient;
import org.eclipse.egit.github.core.service.CommitService;
import org.eclipse.egit.github.core.service.DataService;
import org.eclipse.egit.github.core.service.RepositoryService;
import org.eclipse.egit.github.core.service.UserService;
import org.apache.commons.codec.binary.Base64;

import java.text.SimpleDateFormat;
import java.util.Date;
import java.io.IOException;
import java.util.*;

class GitHubHelper {

    String login;
    String password;

    GitHubHelper( String _login, String _password ) {
        login = _login;
        password = _password;
    }

    public boolean SaveFile( String _repoName,
                             String _title,
                             String _post,
                             String _commitMessage ) throws IOException {
        post = _post;
        repoName = _repoName;
        title = _title;
        commitMessage = _commitMessage;

        boolean rv = false;

        generateContent();
        createServices();
        retrieveBaseSha();

        if( null != baseCommitSha && "" != baseCommitSha ) {
            createBlob();
            generateTree();
            createCommitUser();
            createCommit();
            createResource();
            updateMasterResource();
            rv = true;
        }
```

```
        return rv;
    }

...
```

SaveFile 函数执行使用 GitHub API 把数据写入仓库的各个步骤。后面会一一定义这些函数。可以看出，SaveFile 函数的签名与单元测试中调用的同名函数一样。

下面分别实现 GitHubHelper 类中调用的各个函数。

7.4.3 编写博客内容

首先，实现 generateContent() 函数。下述代码片段中定义的函数用于生成存入 GitHub 仓库里的内容：

```
...
    String commitMessage; // ❶
    String postContentsWithYfm;
    String contentsBase64;
    String filename;
    String post;
    String title;
    String repoName;

    private void generateContent() { // ❷
        postContentsWithYfm =  // ❸
        "---\n" +
        "layout: post\n" +
        "published: true\n" +
        "title: '" + title + "'\n---\n\n" +
        post;
        contentsBase64 =  // ❹
        new String( Base64.encodeBase64( postContentsWithYfm.getBytes() ) );
        filename = getFilename();
    }

    private String getFilename() {
        String titleSub = title.substring( 0,  // ❺
                       post.length() > 30 ?
                       30 :
                       title.length() );
        String jekyllfied = titleSub.toLowerCase() // ❻
        .replaceAll( "\\W+", "-")
        .replaceAll( "\\W+$", "" );
        SimpleDateFormat sdf = new SimpleDateFormat( "yyyy-MM-dd-" ); // ❼
        String prefix = sdf.format( new Date() );
        return "_posts/" + prefix + jekyllfied + ".md"; // ❽
    }

    String blobSha;
    Blob blob;
...
```

可以看出，上述 Java 代码与第 6 章中用于生成文件名和去除空格的 Ruby 代码有很多相似之处。

❶ 首先，声明几个实例变量，用来存储要存入 GitHub 中的数据，包括提交说明、带有 YAML 格式头部元信息的完整文章、编码成 Base64 的文章内容、文件名，以及调用 `SaveFile()` 函数时传入的三个参数，即文章本身、标题和仓库名。

❷ `generateContent` 函数创建文章的各个部分：Base64 编码的全部内容，以及用于存储内容的文件的文件名。

❸ 创建 YAML 格式的头部元信息（YFM 的详细说明参见第 6 章）。我们把布局设为 `post`，把发布与否设为 `true`。YFM 后面要加两个换行。

❹ 使用 Apache Commons 库提供的实用类把博客文章的内容编码成 Base64 格式。Git 仓库里存储的内容使用 UTF-8 或 Base64 格式，因为这是文本内容，所以也可以使用 UTF-8 格式，不过 Base64 是无损的，无需担心内容损坏。

❺ 接下来，在 `getFilename()` 函数中使用文章的前 30 个字符创建标题。

❻ 为了满足 Jekyll 对文章标题的格式要求，把标题转换成小写，再把空格替换成连字符。

❼ Jekyll 要求日期的格式为 `yyyy-MM-dd`，所以我们使用 Java 原生的 `SimpleDateFormat` 类创建符合格式的字符串。

❽ 最后，把这几部分组合在一起构成文件名，再在前面加上 Jekyll 存储文章的目录 `_posts`。

下面创建把提交存入 GitHub 的必要服务。

7.4.4 GitHub服务

接下来，实现 `createServices()` 函数。我们要实例化好几个服务（对 Git 协议的包装），虽然不会立即都使用，但是存储文件过程中的几个步骤会分别用到。这些服务都在 `createServices` 函数中实例化。

```
...
    RepositoryService repositoryService;
    CommitService commitService;
    DataService dataService;

    private void createServices() throws IOException {
        GitHubClient ghc = new GitHubClient();
        ghc.setCredentials( login, password );
        repositoryService = new RepositoryService( ghc );
        commitService = new CommitService( ghc );
        dataService = new DataService( ghc );
    }

...
```

顺便说一下，这样写可以把 GitHub.com 的端点换成企业版的端点。具体的句法参见附录 A。

7.4.5　从仓库和分支中获取基SHA

现在，实现 `retrieveBaseSha()` 函数。Git 仓库是有向非循环图（Directed Acyclic Graph，DAG），因此图中（几乎）每一个节点都指向另一个提交（如果是合并提交，那么可能是两个提交）。向图中添加内容时，我们需要在图中找到之前的节点，然后附加新节点。`retrieveBaseSha` 函数的作用就是如此：查找最后一个提交的 SHA 散列值，即在树中的地址。为了找到这个地址，应用需要引用仓库，于是我们使用前面实例化的仓库服务获取引用。知道仓库后，还要在正确的分支（通过 `getBranch` 函数确定）中查找。

```
...

    private void createServices() throws IOException {
        GitHubClient ghc = new GitHubClient();
        ghc.setCredentials( login, password );
        repositoryService = new RepositoryService( ghc );
        commitService = new CommitService( ghc );
        dataService = new DataService( ghc );
    }

    Repository repository;
    RepositoryBranch theBranch;
    String baseCommitSha;
    private void retrieveBaseSha() throws IOException {
        // 从Git仓库的当前分支中获取基SHA
        repository =  repositoryService.getRepository(login, repoName);
        theBranch = getBranch();
        baseCommitSha =  theBranch.getCommit().getSha();
    }

    public RepositoryBranch getBranch() throws IOException {
        List<RepositoryBranch> branches =
          repositoryService.getBranches(repository);
        RepositoryBranch master = null;
        // 迭代分支列表,查找gh-pages或master
        for( RepositoryBranch i : branches ) {
            String theName = i.getName().toString();
            if( theName.equalsIgnoreCase("gh-pages") ) {
                theBranch = i;
            }
            else if( theName.equalsIgnoreCase("master") ) {
                master = i;
            }
        }
        if( null == theBranch ) {
            theBranch = master;
        }
        return theBranch;
    }

...
```

基 SHA 十分重要，没有它就不能创建提交并把提交链接到现有的提交图中。在前面定义的 SaveFile() 函数中，如果没有正确获取 SHA 散列值，提交步骤就会终止。

7.4.6 创建blob

Git 仓库中的内容以 blob 的形式存储。createBlob 函数先把内容存储成 blob 对象，然后使用 dataService 将其存入仓库。在调用 dataService.createBlob 函数之前，blob 对象并未真正存入 GitHub。此外要注意，blob 对象不能直接链接到 DAG 中，而要通过树对象和提交对象与 DAG 关联，如下所示：

```
...
    String blobSha;
    Blob blob;
    private void createBlob() throws IOException {
        blob = new Blob();
        blob.setContent(contentsBase64);
        blob.setEncoding(Blob.ENCODING_BASE64);
        blobSha = dataService.createBlob(repository, blob);
    }

...
```

7.4.7 生成树

接下来，实现用于生成树的 generateTree() 函数。树包装 blob 对象，并主要提供指向那个对象的路径。如果与操作系统对比，那么树相当于文件名路径，而 blob 相当于索引节点（inode）。数据服务管理器会使用仓库名和基 SHA 地址（之前获取的）验证这是进入仓库的有效入口。创建树之后，要设定必要的树属性，例如树的类型（blob）和模式（blob），以及 blob 对象的 SHA 和大小。然后，使用数据服务对象把树存入 GitHub。

```
...
    Tree baseTree;
    private void generateTree() throws IOException {
        baseTree = dataService.getTree(repository, baseCommitSha);
        TreeEntry treeEntry = new TreeEntry();
        treeEntry.setPath( filename );
        treeEntry.setMode( TreeEntry.MODE_BLOB );
        treeEntry.setType( TreeEntry.TYPE_BLOB );
        treeEntry.setSha(blobSha);
        treeEntry.setSize(blob.getContent().length());
        Collection<TreeEntry> entries = new ArrayList<TreeEntry>();
        entries.add(treeEntry);
        newTree = dataService.createTree( repository, entries,
          baseTree.getSha() );
    }

...
```

7.4.8 创建提交

创建内容的步骤快结束了，接下来要使用 `createCommit()` 函数。我们已经创建了存储内容的 blob，也（算是）创建了存储内容路径的树，不过既然 Git 是版本控制系统，我们还得存储对象创建人和说明。这两个信息存储在提交对象里。做完前面几步之后，应该熟悉这个过程了：创建提交，并添加相关的元数据，即提交说明。此外，还要指明提交用户。然后，使用数据服务在 GitHub 仓库里正确的 SHA 地址处创建提交。

```
...
    CommitUser commitUser;
    private void createCommitUser() throws IOException {
        UserService us = new UserService(); // ❶
        us.getClient().setCredentials( login, password );
        commitUser = new CommitUser(); // ❷
        User user = us.getUser(); // ❸
        commitUser.setDate(new Date());
        String name = user.getName();
        if( null == name || name.isEmpty() ) { // ❹
            name = "Unknown";
        }

        commitUser.setName( name ); // ❺
        String email = user.getEmail();
        if( null == email || email.isEmpty() ) {
            email = "unknown@example.com";
        }
        commitUser.setEmail( email );
    }

    Commit newCommit;
    private void createCommit() throws IOException {
        // 创建提交
        Commit commit = new Commit(); // ❻
        commit.setMessage( commitMessage );
        commit.setAuthor( commitUser); // ❼
        commit.setCommitter( commitUser );
        commit.setTree( newTree );
        List<Commit> listOfCommits = new ArrayList<Commit>(); // ❽
        Commit parentCommit = new Commit();
        parentCommit.setSha(baseCommitSha);
        listOfCommits.add(parentCommit);
        commit.setParents(listOfCommits);
        newCommit = dataService.createCommit(repository, commit); // ❾
    }
...
```

❶ 创建用户服务对象。我们将使用这个对象获取已登录 GitHub 的用户数据。

❷ 然后，创建提交用户。我们将使用这个对象注解提交对象（会用到两次，一次指定作者，一次指定提交者）。

❸ 使用用户服务从 GitHub 中获取用户。

❹ 尝试获取已登录用户的名字。如果没有名字（用户没在 GitHub 个人资料中设置），使用“Unknown”。然后，把名字存入提交用户对象。

❺ 使用相同的方式设定提交用户的电子邮件地址。

❻ 定义 `createCommit` 函数，创建提交对象。

❼ 作者和提交者都要使用，因此传入 `createCommitUser` 函数中创建的提交用户。

❽ 接下来，生成提交列表。我们只会使用一个提交，但是你可能还记得，一个提交可以有多个父提交（例如合并），因此要指定一个或多个父提交。创建提交列表，创建一个父提交，再设置前面得到的基 SHA，然后指明这是新提交的父提交。

❾ 最后，使用数据服务对象创建提交。

7.4.9 更新上游资源

最后，更新分支引用，让它指向新提交的 SHA。

```
...
    TypedResource commitResource;
    private void createResource() {
        commitResource = new TypedResource(); // ❶
        commitResource.setSha(newCommit.getSha());
        commitResource.setType(TypedResource.TYPE_COMMIT);
        commitResource.setUrl(newCommit.getUrl());
    }

    private void updateMasterResource() throws IOException {
        Reference reference =
                dataService.getReference(repository,
                        "heads/" + theBranch.getName() ); // ❷
        reference.setObject(commitResource);
        dataService.editReference(repository, reference, true) ; // ❸
    }
...
```

❶ 首先，创建新提交资源。然后，关联新提交的 SHA，指明这个资源的类型是提交，然后把提交资源与新提交的 URL 链接起来。

❷ 使用数据服务对象从 GitHub 中获取当前分支的引用。为了获取分支引用，要在分支前加上“heads”（前一步已经知道分支）。

❸ 最后，更新分支引用，指向这个新提交资源。

以上是使用 Git Data API 向 GitHub 中添加数据的全部代码。干得漂亮！

7.4.10 通过全部测试

代码写完了，下面要确保测试能通过。

我们要配置测试，让测试在 Android Studio 中运行。点击左侧纵向选项卡里的“Build Variants”，在“Test Artifact”中选择“Unit Tests”。然后，点击“Run”菜单，选择“Edit configurations”。点击加号，选择“JUnit”。此时会显示一个界面，让你创建单元测试的运行配置。首先，点击“Use classpath of module”，选择“app”。确保“Test Kind”选中的是“Class”，然后点击“Class”字段右侧的选择器，会显示出 `GitHubHelperTest.java` 测试类。我们需要将用户名和密码存储为环境变量，所以点击添加这两个环境变量。最终的配置如图 7-5 所示。

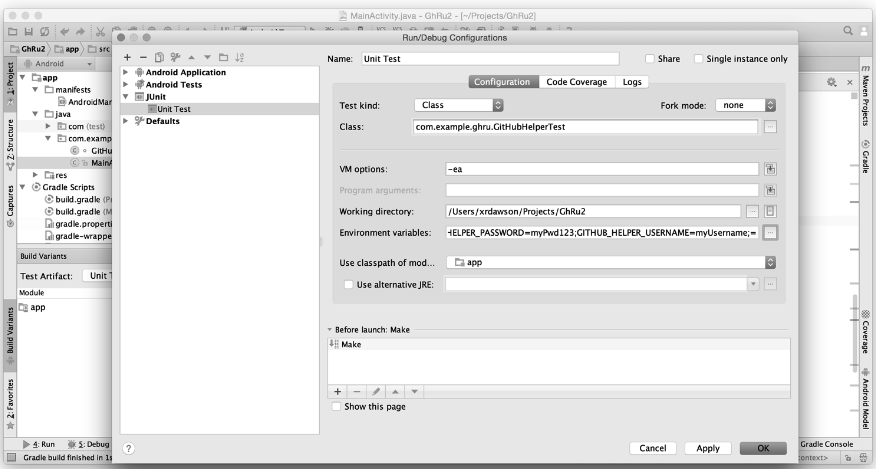


图 7-5：创建单元测试配置

接下来，创建 UI 测试配置。在“Build Variants”标签页中把“Test Artifact”切换成“Android Instrumentation Tests”。然后，点击“Run”菜单，选择“Edit configurations”。点击加号，这一次选择“Android Tests”。在“Module”中选择“app”，然后选择“android.support.test.runner.AndroidJUnitRunner”作为实装运行程序。你可以选择任何设备、仿真器或实体设备（如果有的话）。给配置命名，如“Android Test”。

运行测试的方法是，先切换到正确的测试项目，然后点击“Run”菜单，选择“Debug”，再选择正确的测试配置。我们可以在 Android Studio 中设置断点，逐行运行测试或实现代码。

运行测试时，我发现在不同的构建方式之间切换很麻烦，因此，如果愿意的话，可以使用命令行（这样就不用切换构建方式了）。

```
$ GITHUB_HELPER_USERNAME=MyUsername \
GITHUB_HELPER_PASSWORD=MyPwd123 \
```

```
./gradlew testDebugUnitTest
...
:app:mockableAndroidJar UP-TO-DATE
:app:assembleDebugUnitTest UP-TO-DATE
:app:testDebugUnitTest UP-TO-DATE

BUILD SUCCESSFUL
$ ./gradlew connectedAndroidTest
...
:app:compileDebugAndroidTestNdk UP-TO-DATE
:app:compileDebugAndroidTestSources
:app:preDexDebugAndroidTest
:app:dexDebugAndroidTest
:app:packageDebugAndroidTest
:app:assembleDebugAndroidTest
:app:connectedDebugAndroidTest

BUILD SUCCESSFUL
```

在 Android Studio 的测试运行程序窗口里会看到类似的结果。测试通过了，应用也开发完了。

> 如果想深入学习如何在 Android 中使用 GitHub API，可以看一下 Teddy Hyde（https://github.com/xrd/TeddyHyde.git，Google Play 应用商店中也有）。该应用使用 OAuth 登录 GitHub，为 Jekyll 博客提供了更丰富的编辑功能。

7.5 小结

本章所编写的应用可以发布 Jekyll 博客，添加文章后，GitHub 会自动重新生成网站。这个小应用做的事还挺多的，可以把文件名改为正确的格式，还可以编码提交给 GitHub 的数据。我们还编写了单元测试和 UI 测试，用来验证功能。

下一章将使用 CoffeeScript 创建一个聊天机器人，通过 Activity API 邀请聊天室里的成员审查拉取请求。

第 8 章

CoffeeScript、Hubot和Activity API

GitHub 曾在宣传语中称自己为“社交编程”工具，不过这种表述已经删除了。然而，“社交”仍是 GitHub 所提供服务的核心思想，我们可以使用 Activity API 访问 GitHub 的社交功能。

本章通过实现一个聊天机器人探索 Activity API。你可能觉得奇怪，机器人有违社交精神，为什么要使用机器人呢？其实，我们要开发的是一个合理利用社交 API 的社交机器人。Hubot 是一个易于扩展的聊天机器人，GitHub 用户使用它记录并自动执行任务，还用它在互联网中做些有趣的事情。如果有那么一个机器人能胜任与 GitHub Activity API 的交互，那一定是 Hubot。按照 Hubot 网站（https://hubot.github.com/）的说法，它是一个“能改善生活的可定制机器人”。

8.1 Activity API

Activity API 包含：

- 通知（各种事件中提到用户的评论）
- 加星工具（GitHub 的“加星”相当于 Facebook 的“点赞”，表示认同或感兴趣）
- 关注（跟踪 GitHub 数据的一种方式）
- 事件（高层次的活动流，供用户关注其他用户的活动）

Activity API 也包含订阅源。虽然把订阅源划归给了 Activity API，但是订阅源不能像 API 那样通过编程操作，因此本章不做说明。这里所说的订阅源其实是 Atom 订阅源，交互性不强，是与 RSS 订阅源类似的静态订阅源，可以使用 Atom 客户端订阅。

8.2 让拉取请求得到各方认同

我们将为 Hubot 构建一个扩展。构建好之后，Hubot 将变身能处理下述事项的机器人：

- 使用 GitHub Activity API 订阅通知，监听拉取请求事件；
- 邀请聊天室里的人评论拉取请求；
- 确保机器人与 GitHub 之间的通信是加密传输的（可惜有缺陷）；
- 从外部服务（借助 Slack API）中获取重要信息；
- 自动化测试完全覆盖所有功能；
- 易于模拟 API 和服务的输入输出；
- 便于部署到主流 PaaS（Heroku）。

Hubot 为这个聊天机器人提供了骨架。我们将扩展 Hubot，添加上述功能。从这个过程中你可以看出把这些功能集成到一个机器人中，让它去解决实际问题是多么简单。

8.2.1 注意事项和局限

如果想稳定地运行 Hubot，那就要把它托管到服务器中。Hubot 是使用 NodeJS 开发的，因此要使用支持 NodeJS 的主机服务。我们要通过公网 IP 地址开放 Hubot（不能放在防火墙后面），因为它需要从 GitHub 接收通知。不过，Hubot 不一定非得托管在公开的服务器中，如果 Hubot 不需要从外界接收数据，可以把它托管在私有的内部服务器中。

托管 Hubot 最简单、最便宜的服务是 Heroku。生成 Hubot 之后，只需使用 Git 把它推送到 Heroku 中就能免费发布聊天机器人。本章后文会讲解这些步骤。

Hubot 支持众多聊天端点，可以连接几乎任何流行的聊天服务和协议，包括 IRC、XMPP 以及很多收费服务，例如 Gchat、Basecamp，甚至 Twitter。Slack 是聊天服务领域的新军，虽然初入此行，但是 Slack API 很牢靠，而且可以简单直接地把 Slack 服务与第三方客户端连接起来。我们这个聊天机器人将使用 Slack 作为端点。

下面来创建 Hubot，然后配置，让它使用 Slack。

8.2.2 创建常规的Hubot

搭建 Hubot 机器人之前，要先安装 NodeJS，方法参见附录 B。安装好之后，执行下述命

令，在一个目录中创建 Hubot 的骨架。

```
$ npm install -g generator-hubot ❶
$ mkdir slacker-hubot ❷
$ cd slacker-hubot/
$ yo hubot ❸
$ npm install hubot-slack --save ❹
```

你可能对这些命令不熟悉，下面说明其中几个重要的命令。

❶ `npm` 是为 NodeJS 安装包的工具（参见附录 B）。`npm install -g generator-hubot` 命令安装一个名为 yeoman 的命令行工具和用于生成 Hubot 的 yeoman 插件。

❷ 你应该新建一个目录，然后进入其中，把创建的 Hubot 放在完全独立的地方。

❸ 执行 `yo hubot` 命令，运行生成器，生成 Hubot 机器人所需的最少量的文件。

❹ 然后安装 Slack 适配器，并把包保存到 package.json 文件里。

至此，我们创建了一个简单的 Hubot 机器人，下面该创建 Hubot 赖以生存的 Slack 网站了。

8.2.3 注册Slack账户

创建 Slack 网站的第一步是，访问 https://slack.com/，并注册账户。Slack 网站按组织划分，因此要为 Slack 网站提供 URL 前缀。通常，这是你所在组织的名称。

命名频道

创建好 Slack 网站之后，需要创建一个频道，如图 8-1 所示。

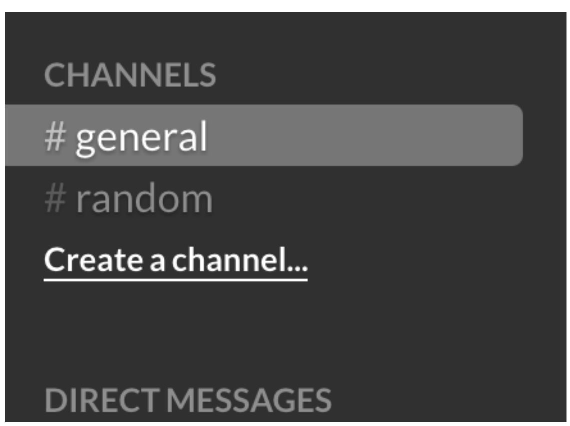


图 8-1：在 Slack 网站的侧边栏中创建一个频道

频道的名字随便取，不过最好起个易记的名字，而且要表明这个频道用于讨论重要事项。你可以使用“PR Discussion”这样的名字，以此表明这个频道是讨论拉取请求的。简单起

见，我们将使用“#general”这个名字。点击创建频道的链接后会弹出一个界面，让你输入名字和可选的描述。创建好频道之后，会看到“Add a service integration”链接，如图 8-2 所示。

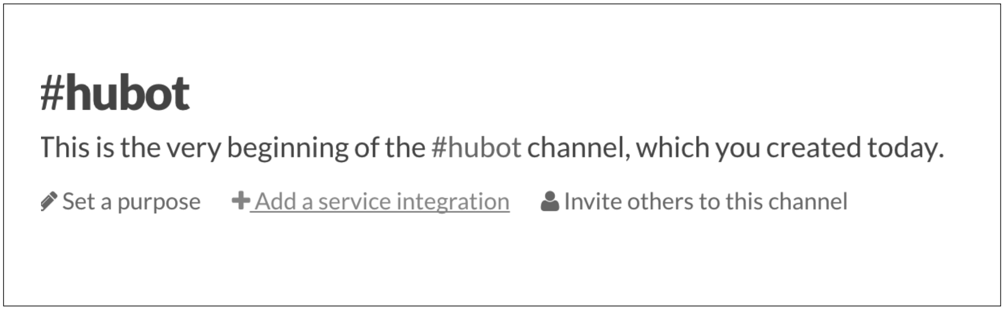


图 8-2：把服务集成到 Slack 中

Slack 支持集成众多服务，其中就有 Hubot，如图 8-3 所示。

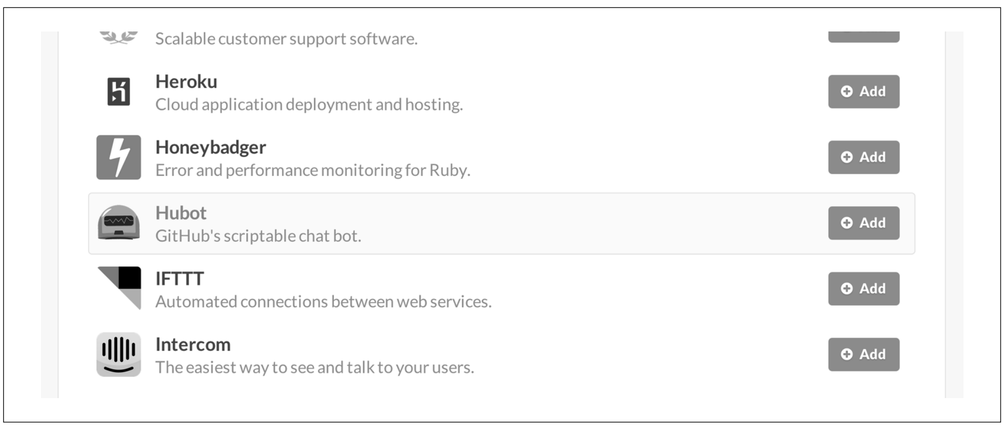


图 8-3：Slack 的服务集成选项

选择 Hubot，此时会进入集成设置界面。

Slack 会自动生成一个身份验证令牌，用于验证与 Hubot 的连接。这个令牌可以吊销，其实图 8-4 中的令牌已经吊销了，Slack 无法再使用它验证身份。如果不小心把这个令牌公开了，可以在这个界面吊销并重新分配令牌。

此外，还要指定名称。我们把名称设为“probot”。如果愿意，可以修改 Hubot 对应的头像。（这些设置如图 8-4 所示。）

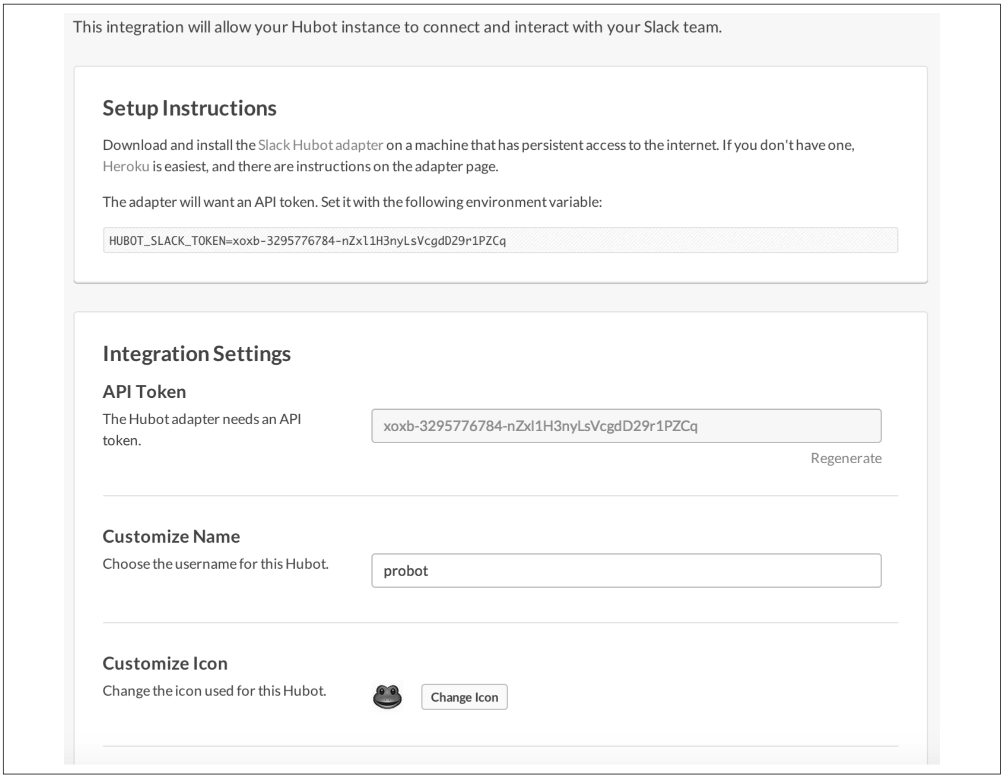


图 8-4：Slack 中配置 Hubot 的页面

继续操作之前记得要保存设置。

8.2.4　在本地运行Hubot

最终，我们将在服务器中运行 Hubot，不过 Hubot 也能在有防火墙保护的笔记本电脑中运行。开发初期，在测试和开发机器人的过程中经常会有改动，此时可能想在本地运行 Hubot。其实，防火墙之后的 Hubot 功能几乎没有损失，唯有一点不足：显而易见，它无法访问外部服务。我们最终的目的是让经过配置的 GitHub 在有拉取请求创建时给我们发送事件，而放在防火墙之后的 Hubot 接收不到这些事件。除此之外，本地运行的 Hubot 几乎所有功能都能用，而且能提升开发的速度。

若想在本地运行，则要在命令行中指定变量。

```
$ HUBOT_SLACK_TOKEN=xoxb-3295776784-nZxl1H3nyLsVcgdD29r1PZCq \
./bin/hubot -a slack
```

这个命令指定使用 Slack 适配器运行 Hubot 脚本。Slack 适配器知道如何与 Slack.com 服务交互。为了提供 Slack 适配器所需的身份验证令牌，我们在命令的开头通过环境变量指定。

1. 首次对话

机器人创建好了，正在 Slack 网站的 #general 房间里待命。进入 #general 房间，输入 Hubot 的名字和命令，例如 `the rules`，确认 Hubot 已经正确连接了。假如 Hubot 的名字是“probot”，应该输入 `probot the rules`。此时，机器人会显示如图 8-5 所示的对话。

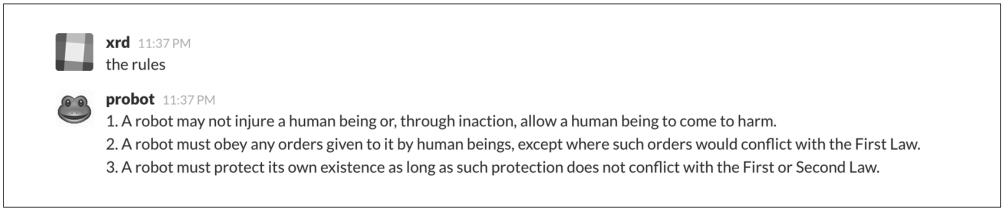


图 8-5：Hubot 内置的应答

我们看到，Hubot 打印出了自己遵守的规则（源自 Isaac Asimov 于 1942 年发表的短篇小说《环舞》）。

2. 了解Hubot支持的命令

Hubot 支持众多原生命令，如果想把这些命令列出来，输入 `help`，如图 8-6 所示。

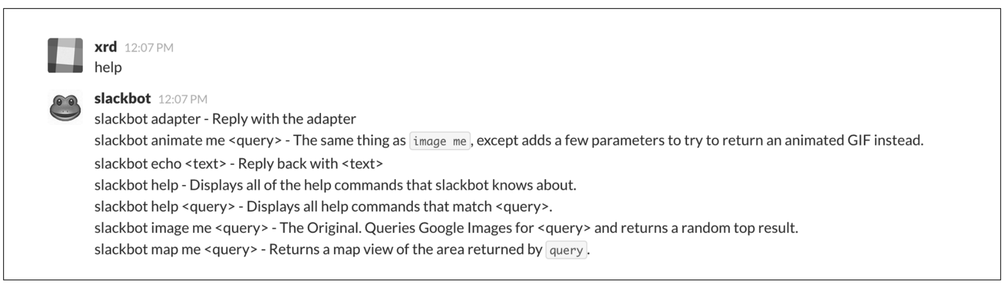


图 8-6：列出 Hubot 支持的命令

`pug me` 命令很受欢迎。很多刚接触 Hubot 的人往往会沉浸几小时，浏览巴哥犬的萌图。别上瘾了！

8.3 部署到Heroku

至此，我们成功地在本地运行了 Hubot。我们可以把机器人部署到 Heroku，让它持续运行，即使关上笔记本电脑也不受影响。

设置Heroku

使用 Heroku 之前要注册账户。Heroku 有免费套餐，我们接下来要做的所有事情使用这个套餐就行。创建账户之后，访问 https://toolbelt.heroku.com/，下载并安装 Heroku 工具集。

这个工具集中有一系列用于管理 Heroku 应用的工具。在此之前，要按照第 1 章所述的方式安装 Ruby。

如果你的聊天机器人能按照前一节所述的指令工作，那么基本上可以部署到 Heroku 了。我们将使用 Heroku 工具集添加与前面一样的环境变量。除了 Slack 的身份验证令牌之外，我们还要配置网站的 URL。Heroku 会根据项目的名称生成 URL（即这里的 `inqry-chatbot`），只要名称没被别人占用，我们就可以自行命名。

```
$ heroku create inqry-chatbot
$ heroku config:add HEROKU_URL=https://inqry-chatbot.herokuapp.com/
$ heroku config:add HUBOT_SLACK_TOKEN=xxbo-3957767284-ZnxlH1n3ysLVgcD2dr1PZ9Cq
$ git push heroku master
Fetching repository, done.
Counting objects: 5, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 317 bytes | 0 bytes/s, done.
Total 3 (delta 2), reused 0 (delta 0)

-----> Node.js app detected
-----> Requested node range:  0.10.x
...
-----> Compressing... done, 6.8MB
-----> Launching... done, v9
       https://inqry-chatbot.herokuapp.com/ deployed to Heroku

To git@heroku.com:inqry-chatbot.git
   d32e2db..3627218  master -> master
```

如果想排除 Hubot 的故障，可以执行 `heroku log` 命令，查看应用的日志，例如 `heroku logs -t`。

```
$ heroku logs -t
2014-11-18T07:07:18.716943+00:00 app[web.1]: Successfully 'connected'
as hubot
2014-11-18T07:07:18.576287+00:00 app[web.1]: Tue, 18 Nov 2014 07:07:18
GMT connect deprecated limit: Restrict request size at location of
read at
node_modules/hubot/.../express/.../connect/.../middleware/multipart.js:86:15
...
```

在聊天室中输入的命令会以事件的形式体现在 Heroku 的日志中。利用这一点，我们可以确认机器人有没有正确连接 Slack。

此外，你可能还想把仓库推送到 GitHub 中。Heroku 除了托管线上应用之外，还会存储 Hubot 机器人完整的 Git 仓库（尽管 Hubot 努力表现，说到底只不过是一个 NodeJS 应用）。Heroku 可以托管 Hubot 机器人的全部源码，但是与 GitHub 相比，缺少一些额外工具，如用户管理。鉴于此，我们应该在 GitHub 中托管源码仓库，这样便于团队成员为聊

天机器人开发新功能。在本地构建并测试之后，我们再使用 Git 工作流程把机器人部署到 Heroku 中。

至此，我们安装了 Hubot，也创建了机器人。下面该探索 Activity API 了，我们要决定如何实现前面计划的扩展。

8.4 Activity API概述

Activity API 的核心是通知，类似于社交网站常见的那种通知，即活动时间线中的重要事件。在 GitHub 中，活动事件通常是指开发者日常的重要操作，例如把提交推送到远程仓库，在仓库的讨论组中提问，或者把工单分配给某个开发者审阅。

团队成员不用通过编程的方式访问 GitHub API，设定规则之后便能收到工作流程中特定事件的邮件通知。用户关注仓库后，有人提交工单、发布评论、发起拉取请求或评论提交时，GitHub 会自动发送通知邮件。此外，即使用户没有关注仓库，当被提到（在评论中使用 @ 加用户名的方式）、被分配工单或者参与仓库的讨论时，也会收到通知。

GitHub 的通知策略过于复杂。很多人严重依赖电子邮件，希望所有重要的活动都能传达给正确的相关人。GitHub 的通知规则丰富，能确保把正确的通知传达给正确的人。

可是，电子邮件的功能已经弱化，变成了待办事项清单；而且，电子邮件虽然易用，但是有时却会导致一个问题：通知被淹没。不停地切换上下文查看邮件时特别容易走神（构建软件时需要极高的注意力），这样容易错过通知。此外，电子邮件是针对私人的，不易协作，人们通常不会共享收件箱中的邮件。下面，我们将扩展 Hubot，把 GitHub 通知发布到共享的通信频道，让用户登陆后可以选择加入，以此来解决上述问题。

8.4.1 编写Hubot扩展

Hubot 扩展使用 JavaScript 或 CoffeeScript 编写。CoffeeScript 是一种中间语言，直接编译成 JavaScript。很多人喜欢使用 CoffeeScript，因为它的句法更简洁，写出的 JavaScript 更安全（CoffeeScript 句法有助于规避 JavaScript 语言常见的隐蔽陷阱，例如 this 引用）。CoffeeScript 是一门缩进式语言（类似于 Python），入门之后，你会发现它比 JavaScript 更易于阅读，嵌套多个回调函数（JavaScript 编程常见的做法）时更能体现这一点，因为通过嵌套等级能轻易区分函数的首尾。Hubot 是使用 CoffeeScript 编写的，我们也将使用 CoffeeScript 编写扩展。

> 对 CoffeeScript 来说，缩进很重要。为了提高可读性，显示从长文件中摘取的代码片段时，我们会修改缩进等级，去掉原始缩进。如果复制时没有重新缩排，代码可能无法运行，因此你要根据上下文重新缩排。

Hubot 扩展的模块机制十分简单。Hubot 会把一个机器人对象传给我们编写的 JavaScript 模块（使用 `export` 句法），把众多 API 方法提供给我们使用。

扩展 Hubot 之前要理解几个概念。scripts 目录里的 example.coffee 文件中有针对各个方法的示例。

- Hubot 有个"大脑"。这是内部状态对象，在不同的聊天消息之间维持着数据。状态默认不存入数据库，因此重启 Hubot 之后状态会消失。不过，我们可以使用 Redis 实现持久化，但这是可选的，而且需要配置。大脑的作用是供我们设定或者获取单个消息中保存的值。
- Hubot 有不同的响应机制。机器人可以在听到完全一致的词语时做出响应，也可以在发现消息中包含关键词时做出响应。不过，我们不用在代码中想方设法区分这两种通信类型。
- Hubot 中有一个 HTTP 服务器。除了聊天服务之外，你可能还想让 Hubot 处理其他服务的请求，而 Hubot 能轻松处理这些请求。
- Hubot 内置了 HTTP 客户端。在 Hubot 中能轻易访问 HTTP 资源，很多流行的 Hubot 扩展在 Hubot 收到请求时会访问 Web 服务。
- Hubot 命令可以包含参数。我们可以编写一个通用函数，通过参数设定让 Hubot 多次执行某件事。
- Hubot 可以处理事件。各个聊天服务都有符合标准 API 的通用事件集。我们可以通过编程的方式让 Hubot 与这些事件交互。例如，变换话题或用户离开房间时可以让 Hubot 执行操作。
- Hubot 可以处理常规的错误。我们可以在代码中编写一个捕获全部错误的处理函数，这样只要代码出错我们就可以捕获，以防机器人崩溃。

我们将使用 Hubot 的前五个功能。

- 使用 Hubot 的大脑存储拉取请求的审查请求。如果 Hubot 邀请某个用户审查拉取请求，那么必须记住用户，这样当用户反馈时，Hubot 才有关于那次邀请的进一步信息。
- Hubot 发出审查邀请后使用 `respond` 方法处理用户接受或拒绝审查请求。
- 使用 HTTP 服务器通过 GitHub 的 webhook 接收拉取请求通知。
- 使用 HTTP 客户端从 Slack 中获取用户列表。
- 使用发给 Hubot 的带参数的请求获取聊天消息中指明的拉取请求 ID。

Hubot 的源码中举例说明了如何使用另外两个功能（事件和常规错误），不过我们的 Hubot 不会用到这些 API。

8.4.2　通过拉取请求审查代码

读过前面的章节可知，拉取请求是 GitHub 采用的一种机制，目的是便于把代码改动集成

到项目中。贡献者可以派生原仓库，然后向那个仓库发起拉取请求；或者，如果贡献者有原仓库的写权限，还可以创建一个 feature 分支，然后向 master 分支发起拉取请求。

拉取请求经常会附带一个消息，指明让哪些人审查。至于该让谁审查，只有创建代码的开发者知道。贡献者可能邀请了正确的人，也可能基于各种（完全合理的）原因，邀请他们喜欢的人审查代码。这或许是让正确的人聚在一起审查新代码的有效方式。

可是，这样邀请审查人员也有缺点：如果被邀请人有其他事，没有看到通知邮件，拉取请求可能长期无人问津。而且，根据一项研究，平均分配任务和责任的团队效率最高。通常，让所有人都参与审查拉取请求中的所有代码效率低下。相反，不让创建代码的人分配，而从项目参与者中随机选择一个开发者审查代码是更好（效率更高）的一种方式。

出现新的拉取请求时，Hubot 会要求聊天室中活跃的用户审查代码。我们将使用 GitHub Activity API 订阅拉取请求事件。Hubot 发现有拉取请求需要审查时，会从聊天室中随机选择一个用户，询问他是否愿意接受挑战。如果用户接受了，我们将在拉取请求的评论中注明。

1. 扩展的样板代码

编写扩展的第一步是定义期望从用户那里接收哪种高级通信格式。我们的目的很简单：寻找表明接受或拒绝审查邀请的回应。扩展脚本应该保存在 scripts 目录中，我们将其命名为 pr-delegator.coffee。这只是我们与用户之间的交互，处理拉取请求通知的代码还没编写。

```
module.exports = (robot) -> ❶
       robot.respond /accept/i, (res) -> ❷
              accept( res )
       robot.respond /decline/i, (res) -> ❸
              decline( res )
       accept = ( res ) -> ❹
              res.reply "Thanks, you got it!"
              console.log "Accepted!" ❺
       decline = ( res ) -> ❻
              res.reply "OK, I'll find someone else"
              console.log "Declined!"
```

代码很多，刚接触 CoffeeScript 的人可能不解其意。读过下述说明之后，希望你能发现这短短一段代码的强大之处。

❶ NodeJS 模块的开头都要使用 `exports` 句法定义入口点。这行代码定义一个接收单个参数的函数。执行这个函数时，参数名为 `robot`。Hubot 框架会传入一个 `robot` 对象供我们使用。

❷ 使用 Hubot API 在 `robot` 对象上定义的 `respond` 方法。这个方法有两个参数：用于匹配的正则表达式和参数为聊天响应对象（这里的 `res`）的函数。后一行把响应对象传给 `accept` 方法。我们稍后定义 `accept` 方法。

❸ 编写函数，处理 `decline` 回应。

❹ 这里定义 `accept` 方法。这个方法的参数是 Hubot 框架生成的响应对象，然后调用 `reply` 方法，把“Thanks, you got it!”消息发送到聊天频道中（你可能猜到了）。

❺ 然后，`accept` 方法调用 `console.log` 方法，把消息输出到启动 Hubot 的控制台中。这是确认一切正常的简单方式，如果没看到这个消息，说明在此之前的代码出错了。频道中的任何用户都看不到 `console.log` 输出的消息。确定生产环境使用的代码后，最好把这行代码删掉；不过，即使忘记，也不会对频道产生任何影响。

❻ 然后，使用与 `accept` 方法同样的 API 定义 `decline` 方法。

如果 Hubot 正在运行中，要重启才能加载扩展脚本。关闭 Hubot（按 Ctrl-C 键），重启，然后在你的 Slack 网站中试试这些命令。输入 `probot accept` 和 `probot decline` 命令，你会看到 Hubot 在频道中做出了响应。此外，在运行 Hubot 的控制台中还会看到 `Accepted!` 或 `Declined!` 消息。

2. 为Hubot扩展编写测试

现在，我们的 Hubot 机器人基本运行起来了，下面要编写测试，以确认代码正确。我们将使用 NodeJS 的测试框架 Jasmine。这个框架使用优雅的行为驱动测试句法，在 `it` 函数的第一个参数中指定行为，然后在第二个参数中定义一个函数，作为测试运行。Jasmine 负责运行各个 `it` 函数，运行结束后会输出细致的报告，指明哪些测试通过了，哪些测试失败了。Jasmine 测试通常使用 JavaScript 编写，不过最新版 Jasmine 还支持使用 CoffeeScript。Hubot 是使用 CoffeeScript 编写的，那么我们也使用 CoffeeScript 编写测试。测试要放在 spec 目录中，而且文件名要以 .spec.coffee 结尾。假设测试文件的完整文件名是 spec/pr-delegator.spec.coffee。Jasmine 期望测试文件的文件名以 .spec 结尾（后面还有扩展名，.js 或 .coffee），否则 Jasmine 不会把文件识别为测试。

```
Probot = require "../scripts/pr-delegator"
Handler = require "../lib/handler"

pr = undefined
robot = undefined

describe "#probot", ->
        beforeEach () ->
                robot = {
                        respond: jasmine.createSpy( 'respond' )
                        router: {
                                post: jasmine.createSpy( 'post' )
                                }
                        }

        it "should verify our calls to respond", (done) ->
                pr = Probot robot
                expect( robot.respond.calls.count() ).toEqual( 2 )
                done()
```

测试的第一行代码把那个 Hubot 扩展模块加载到测试脚本中，我们把函数的返回值存在 Probot 变量中。然后，创建一个 describe 函数，把相关的测试组织在一起。describe 函数的参数是一个标识符（这里的 #probot）和一个包含多个 it 调用的函数。此外，describe 函数中还可以定义 beforeEach 函数，指定各个 it 调用的通用操作。这里，我们创建一个虚拟的 robot 对象，传给后面的 Probot 函数。运行 Hubot 时，Hubot 会创建 robot 对象，并把它传给 Probot 函数；不过在测试中，要生成虚拟的 robot 对象，然后通过方法调用确认它的配置是正确的。如果修改了 Hubot 机器人的代码，而没有更新测试验证改动，那么测试会失败，从而得知需要添加测试或者机器人出问题了。有自动化的健全性检查做保障，我们可以放心地编程，让 Hubot 机器人灵动起来。

你应该发现了，测试也在 robot 对象上调用了几个方法（robot.respond 和 robot.router.post）。我们使用 Jasmine 创建“侦件”（spy），然后生成虚拟函数，记录所有外部（生产环境中的代码或者测试用具）交互。在 it 函数中，我们确认的确调用了那些函数。我们使用 expect 函数验证在 robot 对象上调用了两次 respond 函数，而且还调用了 robot.router.post 方法。

我们需要安装 Jasmine，方法是把它添加到 package.json 文件中。把 "jasmine-node": "^1.14.5" 添加到那个文件中，记得要在前一行末尾添加逗号。添加的代码指定 Jasmine 版本至少为“1.14.5”。

```
...
  "hubot-shipit": "^0.1.1",
  "hubot-slack": "^3.2.1",
  "hubot-youtube": "^0.1.2",
  "jasmine-node": "^2.0.0"
},
"engines": {
...
```

执行下述命令，安装 Jasmine（库本身和测试运行程序的命令行工具），然后运行测试。为了节省空间，此处节略了安装命令的输出。

```
$ npm install
...
hubot-slack@3.2.1 node_modules/hubot-slack
└─ slack-client@1.2.2 (log@1.4.0, coffee-script@1.6.3, ws@0.4.31)
...
$ ./node_modules/.bin/jasmine-node --coffee spec/

.

Finished in 0.009 seconds
1 test, 1 assertions, 0 failures, 0 skipped
```

测试通过了，现在我们有了一种为代码编写文档以及验证代码是否符合预期的方式。

3. 设置webhook

现在该为 Hubot 机器人添加真正的功能了。首先，需要注册拉取请求事件。这个操作可以在 GitHub 网站中执行，不过也可以使用 cURL 工具在命令行中创建 webhook。为此，我们要创建身份验证令牌，然后使用令牌创建 webhook。

执行下述命令创建令牌，记得把用户名变量的值（`xrd`）替换成你自己的。

```
$ export USERNAME=xrd
$ curl https://api.github.com/authorizations --user $USERNAME --data
'{"scopes":["repo"], "note": "Probot access to PRs" }' -X POST
```

上述命令可能有三种响应。如果用户名或密码错误，会得到如下所示的错误响应。

```
{
  "message": "Bad credentials",
  "documentation_url": "https://developer.github.com/v3"
}
```

如果用户名和密码正确，而且没有启用双重身份验证，那么请求成功，返回的 JSON 响应中包含一个令牌。

```
{
  "id": 238749874,
  "url": "https://api.github.com/authorizations/9876533",
  "app": {
    "name": "Probot access to PRs",
    "url": "https://developer.github.com/v3/oauth_authorizations/",
    "client_id": "00000000000000000000"
  },
  "token": "fakedtoken1234",
  "hashed_token": "fakedhashedtoken7654",
  ...
```

如果启用了双重身份验证，会得到如下所示的响应消息。

```
{
  "message": "Must specify two-factor authentication OTP code.",
  "documentation_url":
  "https://developer.github.com/v3/auth#working-with-two-factor-authentication"
}
```

执行上述 cURL 命令后得到的如果是这个消息，你会通过设定的双重身份验证途径（短信、Google Authenticator 等双重身份验证应用或者打印出来的回执码）收到一次性密码。如果设定的是短信途径，查看短信之后再使用 cURL 发送请求，这一次要添加一个首部。

```
$ curl https://api.github.com/authorizations --user $USERNAME --data
'{"scopes":["repo"], "note": "Probot access to PRs" }' -X POST
--header "X-GitHub-OTP: 423584"
Enter host password for user 'xrd':
```

这几步操作成功后（不管有没有使用双重身份验证），你会收到一个 OAuth 令牌。

```
{
  "id": 1234567,
  "url": "https://api.github.com/authorizations/1234567",
  "app": {
    "name": "Probot access to PRs (API)",
    "url": "https://developer.github.com/v3/oauth_authorizations/",
    "client_id": "00000000000000000000"
  },
  "token": "ad5a36c3b7322c4ae8bb9069d4f20fdf2e454266",
  "note": "Probot access to PRs",
  "note_url": null,
  "created_at": "2015-01-13T06:23:53Z",
  "updated_at": "2015-01-13T06:23:53Z",
  "scopes": [
    "notifications"
  ]
}
```

8.4.3 使用OAuth令牌注册事件

现在我们获得了用于创建 webhook 的令牌。执行下述 cURL 命令时记得要使用正确的仓库名和访问令牌。此外，我们还要指定推送到 Heroku 之后获得的端点（这里的 `https://inqry-chatbot.herokuapp.com`）。

```
$ REPOSITORY=testing_repostory
$ TOKEN=ad5a36c3b7322c4ae8bb9069d4f20fdf2e454266
$ WEBHOOK_URL=https://inqry-chatbot.herokuapp.com/pr
$ CONFIG=$(echo '{
  "name": "web",
  "active": true,
  "events": [
    "push",
    "pull_request"
  ],
  "config": {
    "url": "'$WEBHOOK_URL'",
    "content_type": "form",
    "secret" : "XYZABC"
  }
}')
$ curl -H "Authorization: token $TOKEN" \
-H "Content-Type: application/json" -X POST \
-d "$CONFIG" https://api.github.com/repos/$USERNAME/$REPOSITORY/hooks
{
  "url": "https://api.github.com/repos/xrd/testing_repostory/hooks/3846063",
  "test_url":
  "https://api.github.com/repos/xrd/testing_repostory/hooks/3846063/test",
  "ping_url":
  "https://api.github.com/repos/xrd/testing_repostory/hooks/3846063/pings",
  "id": 3846063,
```

```
    "name": "web",
    "active": true,
    "events": [
      "push",
      "pull_request"
    ],
    "config": {
      "url": "https://inqry-chatbot.herokuapp.com/pr",
      "content_type": "json"
    },
    "last_response": {
      "code": null,
      "status": "unused",
      "message": null
    },
    "updated_at": "2015-01-14T06:23:59Z",
    "created_at": "2015-01-14T06:23:59Z"
  }
```

这里用了几个 bash 技巧，但是不难理解。我们创建了几个变量，供最终的命令使用。因为 `$CONFIG` 变量特别长，所以我们使用 `echo` 把中间带有 webhook URL 的一堆信息打印出来了。如果想查看 `$CONFIG` 变量的值，输入 `echo $CONFIG`，你会看到其中一部分是 `... "url": "https://inqry-chatbot.herokuapp.com/pr" ...`，这说明 webhook 的 URL 正确插入了。

这里使用 Heroku API 的 URL 作为 webhook 的端点。这意味着要在 Heroku 中托管应用，webhook 才能与 HTTP 服务器合理交互。我们可以把应用（比如说让 Hubot 连接 Slack 服务）放到防火墙之后，让该应用与聊天室成员交互，不过只有在公开可访问的服务器中运行聊天客户端，webhook 才能成功发送请求。

注意，一定要把 `content_type` 设为 `"form"`（这是默认值，可以留空）。如果把这个字段设为 `json`，那么很难从 Hubot 中获取原始的响应主体，因为接收的是 POST 请求，而且请求会使用安全摘要验证。我们要保证所有请求都是由 GitHub 发出的，不能让恶意的攻击者参与对话。为了防止发生这样的事，我们要使用创建 webhook 时生成的密令验证返回 GitHub 的各个请求。本章后面会深入讨论这个问题，现在要记住的是，创建 webhook 时要设定密令。黑客可能会猜到端点的地址，不过只要没有攻破 Heroku 或 GitHub，他们就不知道 webhook 的密令。

我们要更新测试，确保覆盖这个新功能。我们将使用 Hubot HTTP 服务器，即 Hubot 中运行的内置 Express 服务器。新添加的测试应该验证能在 Hubot 机器人上调用 `router.post` 方法，而且已调用了一次。在测试文件末尾添加下述测试。

```
it "should verify our calls to router.post", (done) ->
        pr = Probot robot
        expect( robot.router.post ).toHaveBeenCalled()
        done()
```

现在运行的话，新添加的测试会失败。我们要在 Hubot 机器人中添加这个功能，处理 GitHub 发送的 webhook 回调。在文件末尾添加下述代码。

```
robot.router.post '/pr', ( req, res ) ->
      console.log "We received a pull request"
```

现在运行测试，会发现全部能通过。通过后，把新版应用发布到 Heroku 中。以后我们会省略这一步，不过要想在设置的路径上收到拉取请求，一定要把文件发布到 Heroku 中，让端点可以公开访问。

```
$ ./node_modules/.bin/jasmine-node --coffee spec/
..
$ git commit -m "Working tests and associated code" -a
...
$ heroku push

Finished in 0.009 seconds
2 tests, 2 assertions, 0 failures, 0 skipped
$ git push heroku master
Fetching repository, done.
Counting objects: 5, done.
Delta compression using up to 8 threads.
...
```

至此，Hubot 机器人的两端都设置好了，接下来该接收 webhook 发送的通知了。

8.4.4　发起真实的拉取请求

现在可以使用真实的 GitHub 通知测试这个 Hubot 机器人了。首先，创建用于测试的仓库。使用第 6 章介绍的 hub 工具可以轻松地在 GitHub 中新建仓库。

```
$ mkdir testing_repository
$ cd testing_repository
$ git init
$ touch test.txt
$ git add .
$ git commit -m "Initial checkin"
$ hub create
...
```

接下来，使用命令行工具为仓库创建一个真实的拉取请求，然后测试 Hubot 机器人。创建拉取请求的常规流程如下：

(1) 新建分支
(2) 添加新内容
(3) 提交内容
(4) 把新分支推送到 GitHub 中
(5) 发起拉取请求

以上这几步可以使用 Git 命令和 cURL 自动执行。有些命令之前用过，我们将利用前面使用 API 通过 cURL 创建 webhook 时用到的变量。配置变量的值跟前面差不多，不过所需的字段不同，这一次我们要设定拉取请求的标题和正文，把 "head" 键对应的值设为分支的名称，还要设置 "base" 键，指明把拉取请求合并到哪个分支。

在试验和学习 Hubot 扩展 API 的过程中可能会多次新建分支、添加内容，再发起拉取请求。这里给出的示例拿来就能用，但是不要天真地以为每次执行的效果都跟第一次一样。因为在试验的过程中可能会多次执行这些命令，所以我们把执行上述操作的命令写入一个 bash 脚本，这些代码是通用的，可以多次运行。我们把这个文件命名为 issue-pull-request.sh，保存在 test 目录里。

```
# 请修改这三个变量
AUTH_TOKEN=b2ac1f43aeb8d73b69754d2fe337de7035ec9df7
USERNAME=xrd
REPOSITORY=test_repository

DATE=$(date "+%s")
NEW_BRANCH=$DATE
git checkout -b $NEW_BRANCH
echo "Adding some content" >> test-$DATE.txt
git commit -m "Adding test file to test branch at $DATE" -a
git push origin $NEW_BRANCH
CONFIG=$(echo '
{ "title": "PR on '$DATE'",
  "body" : "Pull this PR'$DATE'",
  "head": "'$NEW_BRANCH'",
  "base": "master"
}' )
URL=https://api.github.com/repos/$USERNAME/$REPOSITORY/pulls
curl -H "Authorization: token $AUTH_TOKEN" \
-H "Content-Type: application/json" -X POST -d "$CONFIG" "$URL"
```

这个脚本根据当前时间生成一个独一无二的字符串，然后以这个字符串为名创建并检入新分支，再把一些内容添加到一个独一无二的文件里，提交之后推送到 GitHub 中，最后使用 API 生成一个拉取请求。你只需把脚本顶部的三个变量修改为自己的值，而且只需修改一次。这个脚本的适应力很强，如果身份验证令牌是错的（或者过期了），那么它只会把测试数据添加到测试仓库中，因此可以放心试验。不过，要留意 JSON 响应，看请求成功还是失败，如下述代码片段所示。此外，因为我们要把这个脚本当成命令使用，所以要使用 chmod 命令把它设为可执行文件。

现在来运行这个脚本，看看结果如何：

```
$ chmod +x ./issue-pull-request.sh
$ ./issue-pull-request.sh
{
  "url": "https://api.github.com/repos/xrd/testing_repostory/pulls/1",
  "id": 27330198,
```

```
    "html_url": "https://github.com/xrd/testing_repostory/pull/1",
    "diff_url": "https://github.com/xrd/testing_repostory/pull/1.diff",
    "patch_url": "https://github.com/xrd/testing_repostory/pull/1.patch",
    "issue_url": "https://api.github.com/repos/xrd/testing_repostory/issues/1",
    "number": 1,
    "state": "open",
    "locked": false,
    "title": "A PR test",
        "open_issues_count": 1,
  ...
```

返回的 JSON 响应内容很多（这里做了节略），可以看出，第一个字段是拉取请求的链接。不过，`html_url` 字段中的链接才是人类可读的。访问这个链接可以在 GitHub 的网页 UI 中合并拉取请求。

在 GitHub 中查看拉取请求之后，若想深入了解创建拉取请求的过程中发生了什么，可以打开仓库的设置页面，然后点击左侧导航栏里的“Webhooks and Services”，在页面最底部可以看到 webhook 最近发送的请求列表，如图 8-7 所示。

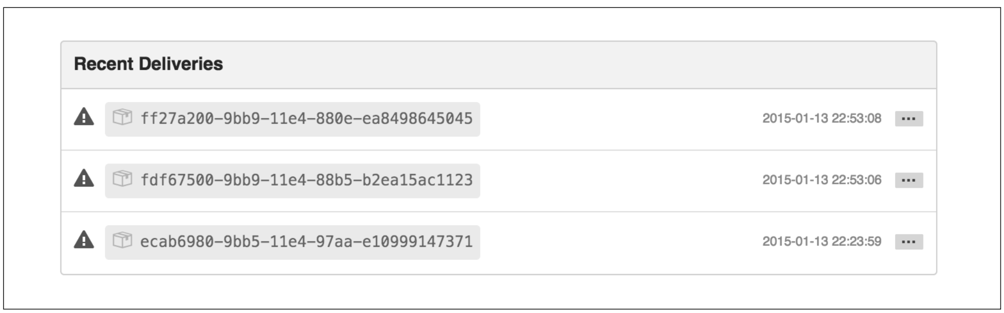


图 8-7：webhook 最近发送失败的请求

这些请求都失败了，因为我们没有正确配置 Hubot 机器人，让它处理 GitHub 发送的真实 HTTP 请求。不过，这却表明 GitHub 收到拉取请求时确实尝试做事了。接下来，我们将编写处理程序的代码，把新代码推送到 Heroku 之后，再发起拉取请求。

8.4.5　通过HTTP POST请求处理拉取请求通知

下面编写 HTTP 处理程序，处理 GitHub 发来的拉取请求通知。起初，你可能会采取简单的方式，直接把这个处理程序添加到顶层脚本中。可是，考虑到 JavaScript 是在回调中处理事件的，而且 Hubot 扩展只导出一个构造方法（使用 `module.exports` 句法），所以更简单也更易于测试的方法是把处理程序写入单独的模块，然后再导入扩展的主脚本。

首先，编写测试。我们已经编写了一个测试，验证了 `robot.router.post` 调用。既然新功能要正式处理拉取请求通知，那就使用 `describe` 句法添加一个新测试组，将其命名为 `"#pr"`。新功能很简单：如果 Hubot 接收到正确的参数（最重要的是内部密令要与请求发

送的密令匹配），那就把拉取请求标识为有效的，然后在聊天室中发布进一步指示，即邀请某个用户审查拉取请求。这个功能的处理程序要导出两个方法，一个是 `prHandler`，把 HTTP 请求发来的信息转发给 `/pr` 路径，另一个是 `setSecret`，用于配置密令。规划好处理程序的内部签名之后，我们先添加两个简单的测试，然后再实现处理程序。

我们要编写两个测试，一个处理正确的流程，另一个处理错误的流程。我们将在 `beforeEach` 块（在每个测试之前都要运行）中创建一个虚拟的 `robot` 对象，并为处理程序模块设置密令。虚拟的 `robot` 对象实现的方法（`messageRoom` 和 `send`）要与 Hubot 提供的真实的 `robot` 对象一样，不过这里将使用 Jasmine 创建侦件，验证实现代码有没有调用这两个方法。

```
describe "#pr", ->
        secret = "ABCDEF"
        robot = undefined
        res = undefined

        beforeEach ->
                robot = {
                        messageRoom: jasmine.createSpy()
                        }
                res = { send: jasmine.createSpy() }
                Handler.setSecret secret

        it "should disallow calls without the secret", (done) ->
                req = {}
                Handler.prHandler( robot, req, res )
                expect( robot.messageRoom ).not.toHaveBeenCalled()
                expect( res.send ).toHaveBeenCalled()
                done()

        it "should allow calls with the secret", (done) ->
                req = { body: { secret: secret } }
                Handler.prHandler( robot, req, res )
                expect( robot.messageRoom ).toHaveBeenCalled()
                expect( res.send ).toHaveBeenCalled()
                done()
```

然后，新建 ./lib/handler.coffee 文件，写入：

```
_SECRET = undefined

exports.prHandler = ( robot, req, res ) ->
        secret = req.body?.secret
        if secret == _SECRET
                console.log "Secret verified, let's notify our channel"
                room = "general"
                robot.messageRoom room, "OMG, GitHub is on my caller-id!?!"
        res.send "OK\n"

exports.setSecret = (secret) ->
        _SECRET = secret
```

可以看出，Hubot API 为我们做了很多事情，它会处理发往 `/pr` 端点的 JSON 格式 POST 请求，把解析得到的参数保存到 `body` 对象中。我们就使用这个对象获取请求中的密令。即使你以前用过 CoffeeScript，可能也不熟悉 `?.` 句法。那行代码的作用是，测试 `body` 是否存在，如果存在，那么获取其中的 `secret` 键。这样一来，即使请求没有发送密令，程序也不会崩溃。如果请求发送的密令与配置的密令一样，那么在聊天室中发布一条消息，否则就忽略请求。不管哪种情况，都要使用 `send` 方法（由 Hubot 用来提供 HTTP 服务器的内部 Express 服务器提供）把响应发给服务器。为了调试，当密令有效时我们输出一个消息，如果不这么做，不管客户端有没有提供密令，响应都是一样的，不易于区分。如果攻击者传入的密令不对，我们不能为他提供任何额外的信息。

现在运行测试，会发现全部能通过。

```
$ node_modules/jasmine-node/bin/jasmine-node \
--coffee spec/pr-delegator.spec.coffee
....

Finished in 0.01 seconds
4 tests, 6 assertions, 0 failures, 0 skipped
```

不管在哪儿运行 Hubot，它都能派生 HTTP 服务器，因此在本地设备中运行时也能与之通信（不过，本地运行的 Hubot 可能在防火墙后面，无法访问 GitHub），于是我们可以在本地使用 cURL 测试。注意，机器人的 `/pr` 端点接收的是 HTTP POST 请求，因此我们要发送 POST 请求（在 cURL 中使用 `--data` 开关）。

```
$ ( HUBOT_SLACK_TOKEN=xoxb-3295776784-nZxl1H3nyLsVcgdD29r1PZCq \
./bin/hubot -a slack 2> /dev/null | grep -i secret & )
$ curl --data '' http://localhost:8080/pr
Invalid secret
OK
$ curl --data 'secret=XYZABC' http://localhost:8080/pr
Secret verified
OK
$ kill `ps a | grep node | grep -v grep | awk -F ' ' '{ print $1 }'`
```

上述命令证实运作正常。首先，启动服务器，通过管道把输出传给 `grep` 命令，输出与密令相关的信息（使用 `&` 符号和括号把整个命令链放在后台运行，这是一个 bash 技巧）。然后，请求本地服务器，我们没有发送密令，因此服务器（运行在同一个 shell 中）调用 `console.log` 方法打印出“Invalid secret”消息；随后，cURL 打印出“OK”，这是服务器返回的响应。然后，再次执行 cURL 命令，不过这一次在 POST 参数中发送了密令。可以看出，Hubot 确认传入的密令与内部保存的密令一样，随后 cURL 再次打印出“OK”，这是 Hubot 中 Express 服务器返回给客户端的响应。最后一行关闭 Hubot：先找到 Hubot 客户端（NodeJS 进程）的 PID，然后向其发送 SIGHUP 信号，关闭 Hubot。

如果正确连接了 Slack 网站，还会在 #general 频道中看到一个消息，即“OMG, GitHub is

on my caller-id!?!”。至此，我们无需手动操作就能发出拉取请求通知了。脚本用于通过 GitHub API 发起真实的拉取请求，而虚拟的 webhook 通知则用于在开发过程中从外部测试脚本中的代码。测试诚然有其作用，但是只使用测试用具做测试而不真正运行 Hubot，那就无法理解 Hubot 内部的运作过程。

1. 分配给聊天室中某个活跃的用户

现在拉取请求通知有了（尽管是我们虚拟的），接下来要编写代码，随机选择一个用户，让他审查拉取请求。

> 这一节是多余的，即使把这一节添加的代码都删掉，我们的 Hubot 机器人也完全能正常运作。撰写本书的过程中，我没意识到 Hubot 的大脑中存储着用户列表，却使用其他方式（Slack API）获取这些数据。使用 Slack API 写完本章后，我才发现自己错了。
>
> 起初，我计划把这一节删掉。可是，这一节却演示了如何使用内置的 HTTP 客户端访问外部服务，这是 Hubot 的一个强大功能，用法很简单。此外，这一节还演示了如何使用强大的测试协助开发 Hubot 扩展。有了测试，我可以重构代码，使用完全不同的方式获取用户列表，而且能确保新代码的处理方式是正确的。此外，Slack API 为登录聊天室的用户提供了更为丰富的数据，虽然这一节用不到，但是在别的场景下却很有用。如果跳过本节，使用前面的编写的代码也行，不过我觉得为了更好地理解 Hubot，本节还是值得一读的。

为了查找聊天室里的用户，可以跳出 Hubot API，使用 Slack API 查询用户列表。Slack API 的一个端点能返回当前在聊天室中的全部用户。我们将使用 Hubot 内置的 HTTP 客户端访问 Slack API。获得用户列表之后，可以随机从中选择一个，邀请他审查拉取请求。

```
_SECRET = undefined

anyoneButProbot = (members) -> ❶
        user = undefined
        while not user
                user = members[ parseInt( Math.random() * \
                     members.length ) ].name
                user = undefined if "probot" == user
        user

sendPrRequest = ( robot, body, room, url ) -> ❷
        parsed = JSON.parse( body )
        user = anyoneButProbot( parsed.members )
        robot.messageRoom room, "#{user}: Hey, want a PR? #{url}"

exports.prHandler = ( robot, req, res ) ->
        slack_users_url =  ❸
                "https://slack.com/api/users.list?token=" +
```

```
            process.env.HUBOT_SLACK_TOKEN
    secret = req.body?.secret ❹
    url = req.body?.url

    if secret == _SECRET and url
            room = "general"
            robot.http( slack_users_url ) ❺
                    .get() (err, response, body) ->
                            sendPrRequest( robot, body, \
                                            room, url ) unless err
    else
            console.log "Invalid secret or no URL specified"
    res.send "OK\n"

exports.setSecret = (secret) ->
    _SECRET = secret
```

❶ 定义一个名为 `anyoneButProbot` 的方法，其参数是用户列表，作用是从中随机选择一个用户，只要不是 Hubot 本身即可。

❷ `sendPrRequest` 方法解析 Slack API 返回的 JSON，把对象里的用户列表传给 `anyoneButProbot` 方法调用。然后，使用 Hubot API 在聊天室中发布一条消息，询问那个选中的用户是否接受拉取请求审查邀请。

❸ 把 Slack API 的令牌添加到 Slack API 的基 URL 后面，构成请求 Slack 服务的 URL。

❹ 跟之前一样，读取密令和拉取请求的 URL，确保二者都存在。

❺ 使用内置的 HTTP 客户端向 Slack API 发起 GET 请求。如果收到的不是错误响应，那么使用 Slack API 提供的数据开始处理拉取请求审查邀请。

可以使用 cURL 命令测试，不过要稍微修改一下：

```
$ curl --data 'secret=XYZABC&url=http://pr/1' \
http://localhost:8080/pr
```

随机选中的用户会看到这条消息：`username: Hey, want a PR? http://pr/1`（Slack 客户端会把 URL 格式化成可点击的链接）。

然而，测试失败，得到这个错误：`TypeError: Object #<Object> has no method 'http'`。这是因为我们在测试中模拟的 `robot` 对象没有 Hubot 内置的 HTTP 接口，因此我们要将其添加到模拟的机器人对象中。HTTP 客户端（由 `node-scoped-http-client` 包提供）的方法签名很复杂：要把几个方法串联起来构成 HTTP 客户端请求，然后把一个回调传给返回的函数，处理响应主体。使用这个模块写出的代码不易于测试（换句话说，我要想方设法弄清楚在测试中该怎么模拟），不过幸好有示例代码，以及说明机器人接口的测试，我这才理解了。我们模拟这个方法链，为模拟的 `robot` 对象定义 `http` 属性。这个属性本身也是函数，调用后返回一个带有 `get` 方法的对象，而调用 `get` 方法返回一个三参数回调函数。这个回调函数中包含错误代码、响应对象和 JSON，但是在这里，只要错误代码是空的，我

们就解析 JSON，获取用户列表，然后发出拉取请求审查邀请。

```
json = '{ "members" : [ { "name" : "bar" } , { "name" : "foo" } ] }'

httpSpy = jasmine.createSpy( 'http' ).and.returnValue(
        { get: () -> ( func ) ->
                func( undefined, undefined, json ) } )

beforeEach ->
        robot = {
                messageRoom: jasmine.createSpy( 'messageRoom' )
                http: httpSpy
                }

        res = { send: jasmine.createSpy( 'send' ) }
        Handler.setSecret secret

it "should disallow calls without the secret", (done) ->
        req = {}
        Handler.prHandler( robot, req, res )
        expect( robot.messageRoom ).not.toHaveBeenCalled()
        expect( httpSpy ).not.toHaveBeenCalled()
        expect( res.send ).toHaveBeenCalled()
        done()

it "should disallow calls without the url", (done) ->
        req = { body: { secret: secret } }
        Handler.prHandler( robot, req, res )
        expect( robot.messageRoom ).not.toHaveBeenCalled()
        expect( httpSpy ).not.toHaveBeenCalled()
        expect( res.send ).toHaveBeenCalled()
        done()

it "should allow calls with the secret", (done) ->
        req = { body: { secret: secret, url: "http://pr/1" } }
        Handler.prHandler( robot, req, res )
        expect( robot.messageRoom ).toHaveBeenCalled()
        expect( httpSpy ).toHaveBeenCalled()
        expect( res.send ).toHaveBeenCalled()
        done()
```

上述测试代码能写出来真是不容易，我重构了多次，确保测试和代码都易于阅读。编写测试要投入很多精力，不过测试和代码都易于阅读而且简洁的话，通常可以确定实现方式是对的。

至此，我们实现了完整可用的代码，这些代码先获取用户列表，然后随机从中选择一个用户，邀请他审查拉取请求。

2. 从Hubot的大脑中获取用户列表

我们可以不用 Slack API，而是调用一个简单很多的方法：`robot.brain.users`。调用 Slack 用户 API 要处理回调，而 `brain.users` 方法不需要，写出的代码更简洁。前面编写的测试

在 Jasmine 创建的 HTTP 侦件上调用了 get 方法，我们要把那部分删掉。然后，为虚拟的 Hubot 机器人的大脑（brain）提供一个新函数，即 users。

可是，不能把代码改成下面这样：

```
...
users = robot.brain.users()
sendPrRequest( robot, users, room, url, number )
...
```

你可能以为 Slack API 返回的用户列表与 Hubot 机器人大脑里存储的用户列表是一样的，其实二者的结构差别很大。那么，我们怎么确定新的结构呢？ NodeJS 的标准库中有个名为 util 的模块，从名称中可以得知，这个模块提供了很多实用的函数。其中一个实用函数是 inspect，其作用是深入对象，以精美的格式把对象的结构打印出来。我们可以使用这个函数和 console.log 查看传给 accept 函数的响应对象，全面了解里面的内容。console.log(require('util').inspect(users)) 这行代码输出的内容如下所示。

```
{ U04FVFE97:
   { id: 'U04FVFE97',
     name: 'ben',
     real_name: 'Ben Straub',
     email_address: 'xxx' },
  U038PNUP2:
   { id: 'U038PNUP2',
     name: 'probot',
     real_name: '',
     email_address: undefined },
  U04624M1A:
   { id: 'U04624M1A',
     name: 'teddyhyde',
     real_name: 'Teddy Hyde',
     email_address: 'xxx' },
  U030YMBJY:
   { id: 'U030YMBJY',
     name: 'xrd',
     real_name: 'Chris Dawson',
     email_address: 'xxx' },
  USLACKBOT:
   { id: 'USLACKBOT',
     name: 'slackbot',
     real_name: 'Slack Bot',
     email_address: null } }
```

嗯，确实如此，Slack API 返回的是数组，而这是关联数组（在其他语言中叫 hash）。因此，我们要重构测试中输入的数据，把数组改成关联数组，然后使用一个函数将其打平（这以后的代码与前面一样了）。该关联数组在调用 robot.brain.users 函数时返回，因此我们要在虚拟的 robot 对象中增加一个侦件，放在 users 键上。

```
...
users = { CDAWSON: { name: "Chris Dawson" }, BSTRAUB: { name: "Ben Straub" } }
brainSpy = {
        users: jasmine.createSpy( 'getUsers' ).and.returnValue( users ),
        set: jasmine.createSpy( 'setBrain' ),
...
```

在实现代码中，要打平用户关联数组，然后在打平后的数组中查找用户。

```
...
flattenUsers = (users) ->
        rv = []
        for x in Object.keys( users )
               rv.push users[x]
        rv

anyoneButProbot = ( users ) ->
        user = undefined
        flattened = flattenUsers( users )
        while not user
                user = flattened[ parseInt( Math.random() * \
                     flattened.length ) ].name
                user = undefined if "probot" == user
        user

...
```

3. 通过webhook发送拉取请求数据

集成就要完成了，接下来要发送真实的拉取请求信息。如果运行 `issue-pull-request.sh` 脚本，会发现它把数据发送给我们的 Hubot 机器人。部署到 Heroku 之后，我们的 Hubot 机器人在公开的主机名上监听。收到拉取请求后，GitHub 会向 Hubot 机器人发送 POST 请求，并在请求主体中发送 JSON 数据。这些数据与 cURL 脚本中使用的 URL 编码参数差别很大，因此要修改代码。

POST 请求发送的 JSON 数据如下所示（为了简洁明了，我重新编排了格式）。

```
{
    "action":"opened",
    "number":13,
    "pull_request": {
      "locked" : false,
      "comments_url" :
      "https://api.github.com/repos/xrd/test_repository/issues/13/comments",
      "url" : "https://api.github.com/repos/xrd/test_repository/pulls/13",
      "html_url" : "https://github.com/xrd/test_repository/pulls/13",
      }
      ...
}
```

其中最重要的是一个 URL（更确切地说是 `html_url`），我们的 Hubot 机器人将在消息中把它发给用户。在 Hubot 机器人中获取并解析这些 JSON 数据的方法很简单。

```
...
exports.prHandler = ( robot, req, res ) ->
        body = req.body
        pr = JSON.parse body if body
        url = pr.pull_request.html_url if pr
        secret = pr.secret if pr

        if secret == _SECRET and url
                room = "general"
...
```

在上述代码中，我们从响应中读取主体内容，然后按照 JSON 格式解析，再从解析后的 JSON 数据中提取密令和那个 URL，然后像之前一样继续处理。

测试也很简单，而且要传入 JSON 数据。

```
...
it "should disallow calls without the secret and url", (done) ->
        req = {}
        Handler.prHandler( robot, req, res )
        expect( robot.messageRoom ).not.toHaveBeenCalled()
        expect( httpSpy ).not.toHaveBeenCalled()
        expect( res.send ).toHaveBeenCalled()
       done()

it "should allow calls with the secret and url", (done) ->
        req = { body: '{ "pull_request" : { "html_url" : "http://pr/1" },
        "secret": "ABCDEF" }' }
        Handler.prHandler( robot, req, res )
        expect( robot.messageRoom ).toHaveBeenCalled()
        expect( httpSpy ).toHaveBeenCalled()
        expect( res.send ).toHaveBeenCalled()
        done()
...
```

方便起见，我们直接在 JSON 数据中指定密令。GitHub 通过 webhook 发给我们的 JSON 数据中没有密令，这么做只是为了简便。现在运行测试，全部都能通过。

4. 保卫webhook

现在，如果发送的密令正确，而且 webhook 发送的数据正确，那么我们的 Hubot 机器人就能正常运作。接下来，我们要保卫 webhook。GitHub 会对 webhook 载荷中的数据签名，从而确认数据的确发自经认可的主机。因此，我们要在处理程序中解码数据。于是，我们要获取 GitHub 在请求首部中发送的安全散列值。然后，使用内部存储的密令计算散列值，如果与首部中的散列值一致，说明入站请求和 JSON 数据确实由 GitHub 发送，而不是由攻击者发送的。

```
...
getSecureHash = (body, secret) ->
        hash = crypto.
```

```
                createHmac( 'sha1', secret ).
                update( "sha1=" + body ).
                digest('hex')
        console.log "Hash: #{hash}"
        hash

exports.prHandler = ( robot, req, res ) ->
        slack_users_url =
                "https://slack.com/api/users.list?token=" +
                process.env.HUBOT_SLACK_TOKEN
        body = req.body
        pr = JSON.parse body if body
        url = pr.pull_request.html_url if pr
        secureHash = getSecureHash( body, _SECRET ) if body
        webhookProvidedHash = req.headers['HTTP_X_HUB_SIGNATURE' ] \
                    if req?.headers
        secureCompare = require 'secure-compare'

        if secureCompare( secureHash, webhookProvidedHash ) and url
                room = "general"
                robot.http( slack_users_url ) ->
                        .get() (err, response, body) ->
                                sendPrRequest( robot, body, \
                                               room, url ) unless err
        else
...
```

GitHub 所做的签名是基于散列函数的报文验证码（Hash Message Authentication Code，HMAC），这种加密方式容易受到时序攻击：在对比计算的散列值和收到的散列值的过程中，攻击者可以暴力获取服务器的访问权。具体到 JavaScript，宽松的比较运算符（如 `==`）就会泄露时序信息。为了防止泄露时序信息，被人用来入侵主机系统，我们要使用 `secure-compare` 模块，在比较散列值时隐藏时序信息。为了加载这个模块，要把它添加到清单文件 package.json 中，然后执行 `npm install secure-compare --save` 命令。

然后，根据处理函数的这个新关注点调整测试。

```
...
it "should disallow calls without the secret and url", (done) ->
        req = {}
        Handler.prHandler( robot, req, res )
        expect( robot.messageRoom ).not.toHaveBeenCalled()
        expect( httpSpy ).not.toHaveBeenCalled()
        expect( res.send ).toHaveBeenCalled()
        done()

it "should allow calls with the secret and url", (done) ->
        req = { body: '{ "pull_request" : { "html_url" : "http://pr/1" }}',
      headers: { "HTTP_X_HUB_SIGNATURE" :
               "cd970490d83c01b678fa9af55f3c7854b5d22918" } }
        Handler.prHandler( robot, req, res )
        expect( robot.messageRoom ).toHaveBeenCalled()
        expect( httpSpy ).toHaveBeenCalled()
```

```
        expect( res.send ).toHaveBeenCalled()
        done()
  ...
```

可以看出，我们把密令从 JSON 数据中移除，放到首部中去。GitHub 的 webhook 会编码 JSON 数据，并在 `HTTP_X_HUB_SIGNATURE` 键中发送安全散列值，因此我们的 Hubot 机器人接收到的也是这种结构。测试中要在模拟的请求对象中提供同样的签名。可以把处理函数中生成安全散列值的代码复制到测试中，不过也可以偷懒，先运行测试（知道这一次会失败），找到 `console.log` 输出的"Hash: cd970490d83c…"，把它复制到模拟的请求对象中。这样做之后，测试就能通过了。

现在，重新加载 Hubot 机器人，然后使用 `issue-pull-request.sh` 脚本发起拉取请求。此时，本该看到一致的散列值，但是却没有（至少使用前述的 package.json 文件时是这样），这是因为写作本书时，Hubot 有个严重的缺陷。

前面说过，Hubot 附带了 NodeJS 平台的一个高性能 Web 框架 Express.js。这个框架采用模块化架构，在请求和响应之间插入了中间件。采用这种方式构建功能，加上数量众多的中间件，Web 开发者可以根据手头要解决的问题，把所需的标准化中间件组件串联起来。常用的中间件有静态文件中间件（用于伺服静态文件）、cookie 中间件、会话中间件和主体解析中间件。可以想象，这些中间件并不是始终都需要（可能还需要其他的）。Express.js 的这种灵活性使得它成为构建 NodeJS Web 应用的首选。

这里，我们特别关注主体解析中间件，使用它把请求的主体转换成 JavaScript 对象，然后依附到请求对象上。前面我们在回调中使用 `req` 变量访问请求对象，显然这个名称是 request 的简写。主体解析中间件能把 HTTP 请求中的任何数据转换成符合 JavaScript 结构要求的关联数组，并将其存储在 `request` 对象中的 `body` 对象中。如果主体是编码的 URL（如果创建 webhook 时把 `content_type` 设为 `form`，拉取请求信息会被编码），主体解析中间件会解码内容，把它当作 JSON 数据解析，然后把得到的对象存储在 `request` 对象的 `body` 属性中。通常，这个过程能省去 Web 应用开发者的大量繁重工作。

可惜，附带的 `express` 对象在加载扩展之前就已经配置好了，我们不能打破加载顺序，无法插入主体解析中间件，因此无法获取原始的主体内容。主体解析中间件处理数据流的方法是，为 HTTP 请求流中的事件注册回调。NodeJS 之所以能在 Web 应用领域占据一席之地，是因为它提供的网络应用工具包大量使用 JavaScript 最具争议的一个特性：异步回调。在 NodeJS 中，进程注册事件之后会把控制权交还宿主程序。在其他语言中，例如 Ruby，如果服务要接收客户端发送的数据，默认情况下要监听入站数据；可是，让程序监听时，其他过程就阻塞了。异步编程并不是什么新概念（例如很多语言都支持多线程），但是 NodeJS 通过事件注册机制简化了与异步函数交互的方式。然而，对 Express.js 的中间件来说，这个事件注册过程却妨碍了我们，因为先加载的中间件先获得入站数据，而等到主体解析中间件处理主体内容时，已经无法获取原始的内容了。我们需要访问原始的主体

内容，但是却无法安装所需的中间件，在我们的 Hubot 扩展收到拉取请求数据时获取原始主体。

那么，该怎么办呢？幸好我们使用的开发栈都是开源的，我们可以修改 Hubot 的代码，根据需求设置 Express 服务器。`npm` 安装的模块存放在 node_modules 目录里，我们可以轻易地在 Hubot 中找到配置 Express 的代码。可是，这么做有问题：如果再次执行 `npm install` 命令，那么 node_modules 目录中的内容全都没有了。在不明确告知的情况下，Heroku 就会这么做。更好的办法是派生 Hubot，把我们的副本存放在 GitHub 中，然后在 package.json? 文件中指定我们派生的副本。然而，这么做也有问题：如果 Hubot 修正了严重的安全漏洞，我们要把改动合并到派生的副本中，这会为维护带来麻烦，不过我们可以使用主仓库中打标签的版本避免这个问题。可见，这个问题没有完美的解决方案，总会导致各种各样的问题。

如果你选择直接修改 Hubot 内部代码，那就要修改 node_modules/hubot/src/ 目录中的 robot.coffee 文件。如果你忘了，我告诉你，node_modules 目录是 NodeJS 包管理器（npm）在本地建立依赖树的目录。robot.coffee 文件是 Hubot 内部构建 `robot` 对象，以及设置 Express HTTP 服务器的地方。我们可以把下述代码添加到第 288 行（如果你使用的 Hubot 版本与我们在 package.json 文件中指定的不同，行号可能不同），这样就能安装所需的中间件，提供验证 HMAC 签名所需的原始主体。

```
...
app.use (req, res, next) =>
  res.setHeader "X-Powered-By", "hubot/#{@name}"
  next()

app.use (req, res, next) =>
  req.rawBody = ''
  req.on 'data', (chunk) ->
    req.rawBody += chunk
  next()

app.use express.basicAuth user, pass if user and pass
app.use express.query()

...
```

Express 中间件的接口十分简单，就是一个 JavaScript 回调函数，参数为请求对象、响应对象和后续要调用的函数。我们在数据内容（主体）传播的过程中注册一个监听器，然后把主体内容赋值给请求对象的一个属性。这样，当请求对象传入 Hubot 机器人处理拉取请求的函数时便能访问原始数据了。`next()` 函数用于告知中间件宿主，可以接着执行下一个中间件了。

接下来要调整测试，满足这个新需求。我们要提供包含 `rawBody` 字段的请求对象，还要使用 `encodeURIComponent` 函数编码内容，让内容的格式与 GitHub 发送的保持一致。

```
...
it "should allow calls with the secret and url", (done) ->
        payload =  '{ "pull_request" : { "html_url" : "http://pr/1" } }'
        bodyPayload = "payload=#{encodeURIComponent(payload)}"
        req = { rawBody: bodyPayload,
        headers: { "x-hub-signature" : \
                 "sha1=dc827de09c5b57da3ee54dcfc8c5d09a3d3e6109" } }

        Handler.prHandler( robot, req, res )
        expect( robot.messageRoom ).toHaveBeenCalled()
        expect( httpSpy ).toHaveBeenCalled()
        expect( res.send ).toHaveBeenCalled()
        done()
...
```

现在，实现代码会导致测试失败，因此我们要修改代码，从请求对象上获取 `rawBody` 属性，把载荷键值对拆分开，然后解码 URI。如果这几步能顺利执行，那么接下来解析 JSON 数据，然后开始验证过程。所有这些在测试中都有描述。修改后的 `prHandler` 方法如下所示。

```
...
exports.prHandler = ( robot, req, res ) ->

        rawBody = req.rawBody
        body = rawBody.split( '=' ) if rawBody
        payloadData = body[1] if body and body.length == 2
        if payloadData
                decodedJson = decodeURIComponent payloadData
                pr = JSON.parse decodedJson

                if pr and pr.pull_request
                        url = pr.pull_request.html_url
                        secureHash = getSecureHash( rawBody )
                        signatureKey = "x-hub-signature"
                        if req?.headers
                                webhookProvidedHash =
                                        req.headers[ signatureKey ]
                        secureCompare = require 'secure-compare'
                        if url and secureCompare( "sha1=#{secureHash}",
                                        webhookProvidedHash )
                                room = "general"
                                users = robot.brain.users()
                                sendPrRequest( robot, users, room, url )
                        else
                                console.log "Invalid secret or no URL specified"
                else
                        console.log "No pull request in here"

        res.send "OK\n"

_GITHUB = undefined

...
```

说了这么多，做了这么多，回头想想，费这么大劲儿验证签名值得吗？如果托管 Hubot 机器人的服务不通过 HTTPS 处理请求，那么验证 HMAC 的过程还是可能会被破解。而且，为了在 Hubot 扩展中进行验证，我们要自己维护一份 Hubot 代码，这样太麻烦了，或许直接忽略验证首部更好。更糟糕的是，现在编写的这个扩展能被攻击者利用来伪造拉取请求通知，并欺骗聊天室里的用户。如果攻击者使用的拉取请求是伪造的，这可能会迷惑 Hubot 机器人，但不会造成真正的破坏。如果使用的是现有的真实拉取请求，攻击者可以迷惑 Hubot 机器人，让它把数据添加到拉取请求中，在评论中乱说有人接受了审查邀请。我们不会在这个扩展中解决这些潜在的问题，不过你可以自己思考如何在 Hubot 机器人中添加代码解决这些问题（例如，检查拉取请求是否已经分配给某个用户了，如果是的话，那就忽略 webhook 后续发来的与那个拉取请求有关的通知）。

5. 回应拉取请求邀请

经过编程，我们的 Hubot 机器人已经可以生成拉取请求审查消息，并随机邀请一个用户了。用户做出回应后会发生什么呢？显然，用户可以做出两种回应：接受邀请，或者拒绝邀请。我们在 Hubot 扩展中放置了占位符，当用户做出回应时会显示一个调试消息，通知我们有用户回应了，然后再给回应的用户发送一个消息。不过，现在我们要处理回应，根据用户的回应在 GitHub 中的拉取请求里标明（假设用户接受了邀请）。

Hubot 有多种方式处理聊天室中的消息，我们选择使用 `respond` 方法。除此之外，也可以使用 `hear` 方法。当消息中包含 Hubot 机器人的名字时，`respond` 方法才会处理；也就是说，Hubot 只会处理 `probot: accept`、`@probot decline` 或 `/ accept`（前提是为 Hubot 名字创建了别名）这样的消息。这里不能使用 `hear` 方法，因为我们要处理的响应很简单，而且消息没有明确的流向。如果使用 `hear` 方法，那么无法保证在正确的上下文中解释消息。因此，使用 `respond` 方法更合理。

如果用户拒绝邀请，那么就发送一条礼貌的消息，注明邀请被拒了：

```
...
exports.decline = ( res ) ->
        res.reply "No problem, we'll go through this PR in a bug scrub"

...
```

我们想邀请一个用户审查拉取请求，但是有可能在较短的时间内出现两个用户。鉴于此，在与目标用户通信的过程中最好标明拉取请求的标识符。此外，还要告知用户，让他们回复 `accept 112` 这样的消息。如此，Hubot 收到消息之后才能解释出用户的意思是接受 112 号拉取请求的审查邀请，而不是 10 秒钟后 John 回应的另一个拉取请求。

如果想这么做，那么 Hubot 机器人要保存拉取请求邀请的状态。幸好，使用 Hubot 的“大脑”能轻松地做到这一点。Hubot 的大脑是一种持久化存储，背后使用的通常是 Redis，我们可以把任何类型的信息存入其中。我们只需使用 `robot.brain` 引用大脑，然后调用 `get`

或 set 方法获取或存储信息。set 方法接受任何键值，不过要注意，如果值为复杂的对象，那么 Hubot 的大脑则无能为力。如果想把复杂的对象序列化成扁平值，那么或许应该调用 JSON.stringify 方法处理对象，确保信息在存取的过程中不损坏。

下面来修改 Hubot 处理程序（并修改扩展文件，处理这个新接口），处理接受和拒绝两种回应。当然，测试也要覆盖这个功能。最后，我们要找到一种方法，把 GitHub API 密钥提供给 Hubot 处理函数。为此，我们将定义一个与设置密令几乎一样的方法。

我们要使用为 NodeJS 开发的 GitHub API 模块，名为 node-github，在 GitHub 中的地址是 https://github.com/mikedeboer/node-github。阅读这个模块的 API 文档后得知，它支持使用 OAuth 令牌验证身份（使用 github.authenticate({ 'type' : 'oauth': 'token' : '...' } 句法），而且提供了一些方法（github.issues.createComment 方法），用于为与仓库关联的工单和拉取请求添加评论。

这两个方法代我们做了大部分工作，知道这一点之后，就可以开始编写测试了。我们将新建一个 describe 块，将其命名为 #response，这个功能的测试都放在这里。前面说过，Hubot 机器人要处理接受和拒绝两种回应，因此这两种代码路径在测试中都要覆盖。这两种情况的设置代码（beforeEach 部分）一样，要向一个随机用户发送拉取请求审查邀请（这个过程在 prHandler 函数中处理）。我们无需再验证这个方法的行为，因为前面的测试已经覆盖了。为处理程序提供所需的状态后，我们要测试 accept 和 decline 方法（处理程序中还没有这两个方法，后面再定义）能正确运作。

接受邀请的处理函数要让 Hubot 机器人联系 GitHub，在拉取请求中添加一个评论，注明聊天室中的某个用户接受了邀请。我们将使用 node-github 模块中绑定的 GitHub API 方法与 GitHub API 通信。这个过程要能测试，因此应该把 GitHub 绑定对象传入接口，然后在测试时传入模拟对象。阅读 GitHub API 绑定中 createComment 方法的文档后得知，它需要关于仓库的信息，例如仓库所属的用户或组织、仓库名、工单号（拉取请求也通过工单号引用）和评论本身。为了获取这些信息，我们只需在接收拉取请求信息的函数中解码收到的信息。这部分代码后面添加（为了测试，要从模块中导出）。我们知道，拉取请求是通过一个大型 JSON 响应传入的，可以使用前面用过的 URL 解码这些信息。因此，#response 块中还要编写两个测试，一个用于测试解码 URL 得到消息对象，另一个用于测试从仓库的拉取请求中存储的评论中获取用户名。我们知道测试 URL 的结构，因为在 webhook 发送的拉取请求消息中见过，但是我们不知道聊天消息（从中提取用户名）的结构，因此知道消息的结构后还要调整测试。

如果用户拒绝要求，那么无需任何操作。从 node-github 模块的文档中可知，在模拟的 GitHub API 绑定中，用户接受邀请后，要登录（使用 authenticate 方法），然后调用 createComment 方法。最后，应该在响应对象上调用 reply 方法，在聊天室中记录操作的结果。

```
...
describe "#response", ->
        createComment = jasmine.createSpy( 'createComment' ).and.
                callFake( ( msg, cb ) -> cb( false, "some data" ) )
        issues = { createComment: createComment }
        authenticate = jasmine.createSpy( 'ghAuthenticate' )
        responder = { reply: jasmine.createSpy( 'reply' ),
        send: jasmine.createSpy( 'send' ) }
        beforeEach ->
                githubBinding = { authenticate: authenticate, \
                                                 issues: issues }
                github = Handler.setApiToken( githubBinding, \
                                                      "ABCDEF" )
                req = { body: '{ "pull_request" : \
                         { url : "http://pr/1" } }', \
                         headers: { "HTTP_X_HUB_SIGNATURE" : \
                         "cd970490d83c01b678fa9af55f3c7854b5d22918" } }
                Handler.prHandler( robot, req, responder )

        it "should tag the PR on GitHub if the user accepts", (done) ->
                Handler.accept( responder )
                expect( authenticate ).toHaveBeenCalled()
                expect( createComment ).toHaveBeenCalled()
                expect( responder.reply ).toHaveBeenCalled()
                done()

        it "should not tag the PR on GitHub if the user declines", \
               (done) ->
                Handler.decline( responder )
                expect( authenticate ).toHaveBeenCalled()
                expect( createComment ).not.toHaveBeenCalledWith()
                expect( responder.reply ).toHaveBeenCalled()
                done()

        it "should decode the URL into a proper message object " + \
                "for the createMessage call", (done) ->
                url = "https://github.com/xrd/testing_repository/pull/1"
                msg = Handler.decodePullRequest( url )
                expect( msg.user ).toEqual( "xrd" )
                expect( msg.repository ).toEqual( "testing_repository" )
                expect( msg.number ).toEqual( "1" )
                done()

        it "should get the username from the response object", (done) ->
                res = { username: { name: "Chris Dawson" } }
                expect( Handler.getUsernameFromResponse( res ) ).toEqual \
                                                        "Chris Dawson"
                done()
```

注意，为了节省空间，我对这段代码做了适当的缩进。你写出的代码可能要多缩进几层。如果有疑问，请参考示例仓库中具体的代码。

如果现在运行，那么测试会失败。接下来要在扩展的末尾编写代码。我们要解析 URL，将

其转换成结构适当的消息对象；要把提醒发到 GitHub 中拉取请求的评论里；还要从传给我们的响应对象中提取用户。前两点易于实现，阅读 GitHub API 绑定的文档后，使用基本的 JavaScript 就能写出代码。第三点需要深入研究一下，我们暂且不实现，只留个占位符。

为了把 URL 转换成 `createMessage` 方法所需的对象，要在斜线处分拆 URL，然后通过索引获取所需的元素。我们或许还要添加一些测试，覆盖传入空字符串的情况，以及其他边界情况；不过这些留作练习，交给读者去做。遇到这些情况时，我们的代码不会出错，但是在测试中最好覆盖。

```
...
_GITHUB = undefined
_PR_URL = undefined

exports.decodePullRequest = (url) ->
        rv = {}
        if url
                chunks = url.split "/"
                if chunks.length == 7
                        rv.user = chunks[3]
                        rv.repository = chunks[4]
                        rv.number = chunks[6]
        rv

exports.getUsernameFromResponse = ( res ) ->
        "username"

exports.accept = ( res ) ->

        msg = exports.decodePullRequest( _PR_URL )
        username = exports.getUsernameFromResponse( res )
        msg.body = "@#{username} will review this (via Probot)."

        _GITHUB.issues.createComment msg, ( err, data ) ->
                unless err
                        res.reply "Thanks, I've noted that in a PR comment!"
                else
                        res.reply "Something went wrong, " + \
                                  "I could not tag you on the PR comment."

exports.decline = ( res ) ->
        res.reply "OK, I'll find someone else."
        console.log "Declined!"

exports.setApiToken = (github, token) ->
        _API_TOKEN = token
        _GITHUB = github
        _GITHUB.authenticate type: "oauth", token: token

exports.setSecret = (secret) ->
        _SECRET = secret
```

简单来说，这里添加了一个内部变量，名为 `_GITHUB`，让它引用 GitHub API 绑定实例。接

口中的 `setApiToken` 方法有个参数就是这个实例。`setApiToken` 方法的参数是 OAuth 令牌和绑定，这样便于在测试中传入模拟的绑定。在测试之外运行时，这个方法会在 GitHub API 绑定上调用 `authenticate` 方法，为使用 API 绑定连接 GitHub API 做好准备。

现在，这个扩展的顶层脚本如下所示。

```
handler = require '../lib/handler'

handler.setSecret "XYZABC"
github = require 'node-github'
handler.setApiToken github, "12345ABCDEF"

module.exports = (robot) ->
        robot.respond /accept/i, ( res ) ->
                handler.accept( res )

        robot.respond /decline/i, ( res ) ->
                handler.decline( res )

        robot.router.post '/pr', ( req, res ) ->
                handler.prHandler( robot, req, res )
```

从上述代码可以看出，接口简洁明了，这是因为主要工作都在经过完好测试的处理程序中处理。

6. 查看响应对象

我们要获取用户名，而有理由相信传给我们的响应对象中有用户名。Hubot API 中的 `respond` 方法的说明文档大部分是一些 Hubot 示例脚本，基本不知道把参数传给回调会怎样。下面使用 `util` 库查看数据，把数据打印到控制台中。这里对完整的输出做了节略，从中可以看出，里面包含发消息给 Hubot 机器人的用户相关的信息。如果想获取用户的名字，可以使用 `response.message.user.name`。

```
{ robot:
   { name: 'probot',
     brain:
      { data: [Object],
  ...
  message:
   { user:
      { id: '...',
        name: 'xrd',
        real_name: 'Chris Dawson',
        email: 'chrisdawson@example.com'
     ...
     text: 'probot accept',
     rawText: 'accept',
     rawMessage:
      { _client: [Object],
  ...
```

```
    match: [ 'probot accept', index: 0, input: 'probot accept' ],
    ...
  }
```

我们可以从中获取所需的全部信息，尤其是用户名和电子邮件地址。那么，下面来更新测试和处理程序的代码。把测试文件中的最后一个测试改成下面这样：

```
...
it "should get the username from the response object", (done) ->
        res = { message: { user: { name: "Chris Dawson" } } }
        expect( Handler.getUsernameFromResponse( res ) ).toEqual "Chris Dawson"
        done()

...
```

而处理程序中定义的 `getUsernameFromResponse` 方法要改成：

```
...
exports.getUsernameFromResponse = ( res ) ->
        res.message.user.name

...
```

有了这些信息之后，就可以在拉取请求中发布评论了。好吧，还差一点儿。

7. 通过Collaborators API统一用户名

如果接受拉取请求邀请的用户在 Slack 中的用户名与 GitHub 中的用户名完全一致，那么可以假定确有其人，然后在拉取请求中发布评论，提醒该用户（以及其他用户）这个拉取请求由他审查。可以使用 Repository API 中的协作者分项查找用户的 GitHub 用户名。

如果在协作者列表中找不到用户，没有与 Slack 用户名完全一样的名字，这说明我们至少有一个问题（也许有两个问题）。首先，我们可能错误匹配了他们的身份（他们在每个网站的用户名不同）。如果是这样，可以在 Slack 网站的聊天室里请用户澄清该问题。此外，还有另外一种情况：接受邀请的用户不是 GitHub 仓库的协作者。如果是这样，即使问清用户名也没用。Repository API 支持添加协作者，因此这里可以这样做；但是，可能需要经过一番讨论之后才能决定要不要这么做（获取仓库的写权限后要担起很大的责任，不像参与聊天室那么轻松）。因此，不能在聊天室中通过自动化的方式把用户添加为仓库的协作者。鉴于此，我们只会编写代码、统一聊天室和 GitHub 中的用户名，而不会处理用户不是仓库协作者的情况。

我们可以使用传入 `setApiToken` 方法的 GitHub API 绑定验证用户是不是仓库的协作者。因此，要使用 API 绑定在 `repos` 命名空间中提供的 `getCollaborator` 方法。这个方法的第一个参数是一个消息，用于指定仓库和属主，消息的 `collabuser` 属性用于确保用户是协作者；第二个参数是一个回调，在请求完成后执行。如果回调没有返回错误，Hubot 机器人应该发布一条确认评论，并在聊天室中发布消息。

即将添加的测试要验证 `repos.getCollaborator` 调用的行为。我们要在测试的设置块中模拟 `getCollaborator` 调用，然后使用 Jasmine 创建侦件，这样便能在测试中确认 `getCollaborator` 被调用了。设置代码比之前都长，不过采用的模式不变，还是生成侦件，监视方法，再根据需求实现虚拟的回调。此外，还可以把响应对象里的消息移到设置块中，这样每个子测试都能使用，而无需在每个测试主体中创建。

```
...
send: jasmine.createSpy( 'send' ),
message: { user: { name: "Chris Dawson" } } }
getCollaborator = jasmine.createSpy( 'getCollaborator' ).and.
        callFake( ( msg, cb ) -> cb( false, true ) )
repos = { getCollaborator: getCollaborator }

...

it "should tag the PR on GitHub if the user accepts", (done) ->
        Handler.accept( robot, responder )
        expect( authenticate ).toHaveBeenCalled()
        expect( createComment ).toHaveBeenCalled()
        expect( responder.reply ).toHaveBeenCalled()
        expect( repos.getCollaborator ).toHaveBeenCalled()
        done()
```

现在，处理程序可以完整实现 `accept` 和 `decline` 方法了。

```
...
exports.accept = ( robot, res ) ->

        prNumber = res.match[1]
        url = robot.brain.get( prNumber )

        msg = exports.decodePullRequest( url )
        username = exports.getUsernameFromResponse( res )
        msg.collabuser = username

        _GITHUB.repos.getCollaborator msg, ( err, collaborator ) ->
                msg.body = "@#{username} will review this (via Probot)."

                _GITHUB.issues.createComment msg, ( err, data ) ->
                        unless err
                                res.reply "Thanks, I've noted that " + \
                                          "in a PR comment. " + \
                                          "Review the PR here: #{url}"
                        else
                                res.reply "Something went wrong."  + \
                                          "I could not tag you " + \
                                          "on the PR comment: " +
                    "#{require('util').inspect( err )}"

exports.decline = ( res ) ->
        res.reply "No problem, we'll go through this PR in a bug scrub"
...
```

至此，我们在 Hubot 机器人中完整实现了 `accept` 和 `decline` 这两个方法。

8. 去掉源码中的机密信息

通常，我们不能在源码中保存密码（或其他访问凭据，如 OAuth 令牌或密令）。而目前我们都直接把这些信息写在 pr-delegator.coffee 文件中了。机密信息可以从进程的环境变量中获取。

```
...
handler.setSecret process.env.PROBOT_SECRET
github = require 'github'
ginst = new github version: '3.0.0'
handler.setApiToken ginst, process.env.PROBOT_API_TOKEN
...
```

在命令行中启动 Hubot 机器人时，要像在本地笔记本电脑中测试那样调用命令。

```
$ PROBOT_SECRET=XYZABC \
PROBOT_API_TOKEN=926a701550d4dfae93250dbdc068cce887531 \
HUBOT_SLACK_TOKEN=xoxb-3295776784-nZxl1H3nyLsVcgdD29r1PZCq \
./bin/hubot -a slack
```

发布到 Heroku 之后，要使用相应的 Heroku 命令设置这些环境变量。

```
$ heroku config:set PROBOT_API_TOKEN=926a701550d4dfae93250dbdc068cce887531
Adding config vars and restarting myapp... done, v12
PROBOT_API_TOKEN=926a701550d4dfae93250dbdc068cce887531

$ heroku config:set PROBOT_SECRET=XYZABC
Adding config vars and restarting myapp... done, v12
PROBOT_SECRET=XYZABC
```

别忘了，运行测试时也要在命令行中指定这些环境变量。

```
$ PROBOT_SECRET=XYZABC \
PROBOT_API_TOKEN=926a701550d4dfae93250dbdc068cce887531 \
node_modules/jasmine-node/bin/jasmine-node --coffee \
spec/pr-delegator.spec.coffee
```

8.5 小结

我们的 Hubot 机器人运转起来了！本章从头开始构建了一个机器人，让它与聊天室中的成员互动。然后，我们重构了机器人，把代码写入不同的模块，让功能易于测试。在这个过程中，我们熟悉了 Hubot API，甚至还讨论了如何修改 Hubot 的源码（以及随之而来的弊端）。最后，我们演示了如何使用 Activity API 接收 GitHub 通过 webhook 发送的数据（包括虚拟数据）。

下一章将构建一个单页应用，使用 JavaScript 和 GitHub.js 库与 Pull Request API 交互，编辑 GitHub 仓库中的信息。

第 9 章

JavaScript和Git Data API

利用 GitHub API 开发的应用通常放在服务器中，然而，并不是说只能使用服务器端编程语言访问 GitHub API。GitHub API 也能在 Web 浏览器中使用，如果知道一点 HTML，应用的 UI 也随之有了。本章讨论如何使用非官方的 JavaScript 客户端库访问 GitHub API，构建一个单页应用（Single-page Application，SPA），而且完全托管在 GitHub 中。

JavaScript 的主要缺点是不易测试。JavaScript 的一大特色是异步，这正是导致测试难以编写的主要原因。直到最近出现回调轮询，测试非线性代码才变得简单。不过，近期出现的工具，如 AngularJS 和基于 promise 的库，不仅易于测试，而且还简明。使用第三方服务构建应用更应该测试，本章在开发应用的过程中会添加测试，确认功能是否符合预期。

学过其他命令式编程语言的人通常易于上手 JavaScript。然而，JavaScript 有个难以理解的特性：回调函数。在 JavaScript 中，函数是一等对象，意即函数可以作为参数传给其他函数，也能作为值存储在变量中。JavaScript 编程离不开回调。有时，回调会导致 JavaScript 代码难以调试和理解。如前所述，编写带有测试的代码更易于了解全局，所以本章我们也会编写测试，让起初难以理解的回调函数变得易于理解。

9.1　构建一个咖啡店数据库并托管在GitHub中

与许多软件开发者一样，我几乎离不开咖啡。或许，我们一家人都是如此，每搬到一座新城市，我都会拖着妻儿走街串巷，看哪家店煮的咖啡和做的无麸质点心最好。

此时，Google 地图能帮上大忙，因为它能告诉我哪里有咖啡店，还有评价。不过，Google 地图提供的咖啡店信息不够详细，范围有限。我想知道哪家店提供不含奶的米浆，以及去

某家店之前要了解的具体事项。市面上导航和地图应用很多，但如果它们不符合我的个性化需求，那么我将错过一次绝佳体验。这个问题摆在我面前，急需解决。下面我们就来使用 GitHub API 解决这个问题。

本章将构建一个咖啡店单页应用，允许任何人添加关于咖啡店的信息。这些信息灵活、动态，可以搜索和过滤。这个应用中的所有文件，如 HTML、图像和 JavaScript 都托管在 GitHub 中。而且，我们将使用 GitHub API 来让协作者为数据库添加数据。这个数据库也托管在 GitHub 中。GitHub 开发者都编写带有测试的代码，因此我们也会编写测试验证 JavaScript 代码，确保应用与 GitHub API 的交互符合预期。

具体而言，我们将使用下述技术。

- 一个（非官方）GitHub API JavaScript 库（https://github.com/michael/github）。
- AngularJS（http://angularjs.org），这个框架用于编写 JS 应用，功能特别强大，而且易于测试。
- Bootstrap（http://getbootstrap.com），这是一个 CSS 库，使用它能轻易构建出精美的 Web 应用。

阅读本章不需要特别了解这些技术。

9.2 搭建环境

首先创建应用的主页，然后将其推送到仓库中。

```
$ mkdir coffeete.ch
$ cd coffeete.ch
$ git init
$ git checkout -b gh-pages
$ printf "<html>\n<body>Hello from CoffeeTe.ch</body>\n</html>\n" > index.html
$ git commit -m "Add starting point index.html" -a
$ git config push.default gh-pages
```

注意，这里新建了一个仓库，然后创建并进入了 `gh-pages` 分支。一切操作都将在这个分支里执行。随后使用 `git config` 命令指定默认推送到 `gh-pages` 分支。这样就可以使用 `git push` 推送，代替较长的 `git push origin gh-pages`。

9.2.1 绑定域名

把文件推送到 GitHub 中的仓库之后，可为仓库绑定一个真实域名。这个操作分两步。

- 添加 CNAME 文件，告诉 GitHub 把服务解析到哪个域名。
- 设置 DNS 记录，把域名指向 GitHub 正确的 IP 地址。

假设你的域名是 myspecialhostname.com。如果想把这个仓库绑定到二级域名 coffeetech 上，要像下面这样做。

```
$ echo 'coffeetech.myspecialhostname.com' > CNAME
$ git commit -m "Added CNAME mapping" -a
$ git push
```

注意，要等大约 10 分钟，让 GitHub 重新生成数据库，在前端服务器中把 `gh-pages` 分支中的网站与域名对应起来。只有第一次把仓库和域名连接起来时要等这么久，后续改动几乎都立即生效。

> 一般来说，DNS 记录要等几小时甚至几天时间才能传播开，因此在网站上线之前要提前绑定好域名。

接下来可以安装这个应用所需的库了。

9.2.2 添加支持库

前面说过，我们将使用 GitHub.js 库、AngularJS 和 Bootstrap。下面把这些库添加到项目中。使用你喜欢的编辑器，把 index.html 改成下面这样。

```
<html>
<head>
<title>CoffeeTe.ch</title>
<meta name="viewport" content="width=device-width, initial-scale=1.0"> ❶
<link rel="stylesheet" type="text/css" href="bootstrap.min.css"></link>
</head>
<body ng-app> ❷
<div class="container">
{{'Welcome to Coffeete.ch'}} ❸
</div>
<script src="angular.js"></script>
<script src="github.js"></script>
</body>
</html>
```

此处假设你已掌握了大多数 HTML 概念，不过下面要说几个进阶的话题。

❶ 这个 `meta` 标签让我们的页面能完好适应移动浏览器，并且启用 Bootstrap 的响应式功能。

❷ `body` 标签中的 `ng-app` 属性告诉 AngularJS 初始化并编译自此之后的页面。

❸ `{{ }}`（两对花括号）是 AngularJS 双向数据绑定指令。如果你不熟悉这个概念，稍后就会看到用法。我们编写这行代码的目的是检查 AngularJS 能否正确运行。如果看到不

带花括号的“Welcome to Coffeete.ch”，说明 AngularJS 加载了，可以正常运行。如果看到花括号，说明有问题需要解决。双向数据绑定解决了构建 JS 应用时的一大痛点：整理网络事件中来回发送的数据，写入 HTML 或从 HTML 表单中提取。这些繁复的操作都由 AngularJS 完成。稍后我们会在 AngularJS 作用域中定义一个变量，然后使用 {{ }} 指令访问变量，以此说明如何使用双向数据绑定。

然后，使用下述命令把所需的文件下载到本地。我们要下载 AngularJS、GitHub.js 和 Bootstrap CSS。

```
$ wget https://ajax.googleapis.com/ajax/libs/angularjs/1.2.10/angular.js
$ wget https://maxcdn.bootstrapcdn.com/bootstrap/3.3.5/css/bootstrap.min.css
$ wget https://github.com/michael/github/raw/master/github.js
```

现在可以在这个单页应用中使用那个 GitHub 库了。

9.3 使用GitHub.js开发一个AngularJS应用

下面要编写 coffeetech.js 文件。这个单页应用的功能都在这个文件中实现。在仓库的根目录中新建一个文件，将其命名为 coffeetech.js，然后写入下述代码。

```
var mod = angular.module( 'coffeetech', [] ) ❶
mod.controller( 'GithubCtrl', function( $scope ) { ❷
  var github = new Github({} ); ❸
  var repo = github.getRepo( "gollum", "gollum" ); ❹
  repo.show( function(err, repo) { ❺
    $scope.repo = repo;
    $scope.$apply(); ❻
  });
})
```

❶ 定义一个模块，名为“coffeetech”。把这个模块的引用保存在一个变量中，后面定义控制器（一种小型函数包）时要用。在 AngularJS 中，模块用于把相关的功能组织在一起。这个应用的所有代码都将放在这个模块中。

❷ 定义一个控制器，名为 `GithubCtrl`，用于放置函数和数据。使用这个句法定义控制器时，要为控制器起个名字，然后定义一个至少有一个参数（`scope` 对象）的函数。我把 `scope`（作用域）理解成控制器有权访问的“地域”。控制器只能访问所在 `scope` 中的数据和函数，只要函数和变量在 `scope` 中定义，AngularJS 就能自动做很多事情。

❸ 使用构造方法创建一个 `Github()` 对象。调用这个构造方法时可以指定用户凭据，不过这里不用指定，因为要访问的是公开仓库。

❹ 得到 `github` 对象之后，调用 `getRepo()` 方法，传入仓库属主和名称，获得仓库对象。

❺ 为了加载仓库中的数据，在仓库对象上调用 `show` 方法。传给 `show` 方法的参数是一个回调，它有两个参数 `err` 和 `repo`，前一个用于处理错误，后一个用于提供指定仓库的详

细信息。这里使用公开的 Gollum 维基仓库，显示一些示例数据。

❻ 加载仓库数据之后，要调用 `$apply` 方法，告诉 AngularJS 作用域变量中存储的数据变了。前面说过，AngularJS 知道在 `scope` 中定义的函数和数据，而 `show` 方法在 GitHub 对象上定义，AngularJS 不知道这里的变动，因此要调用 `$apply()` 方法。

GitHub.js 会代我们请求 GitHub，而 AngularJS 负责把结果放入网页。为了让网页使用这个数据，要修改 index.html 文件中的 HTML 如下。

```
<html>
<head>
<title>CoffeeTe.ch</title>
<meta name="viewport" content="width=device-width, initial-scale=1.0">
<link rel="stylesheet" type="text/css" href="bootstrap.min.css"></link>
</head>
<body ng-app="coffeetech"> ❶
<div class="container" ng-controller="GithubCtrl">
{{ repo }} ❷
</div>
<script src="angular.js"></script>
<script src="github.js"></script>
<script src="coffeetech.js"></script> ❸
</body>
</html>
```

❶ 修改 `ng-app` 属性，让它引用 coffeetech.js 文件中定义的模块。

❷ 删掉绑定 `Welcome to CoffeeTech` 字符串的数据，改成对 `repo` 变量的绑定（AngularJS 默认会过滤复杂的对象，把它转换成 JSON）。

❸ 在引入其他 JS 的标签下面引入 coffeetech.js 文件。

如果在浏览器中查看这个页面，会看到如图 9-1 所示的内容。



图 9-1：完全混乱的 JSON

显示的数据好多啊！虽然 AngularJS 的 JSON 过滤器会以精美的格式把数据打印出来，但是这也有点太多了。接下来修改 HTML，去掉一些噪声信息。

```
<html>
<head>
<title>CoffeeTe.ch</title>
<meta name="viewport" content="width=device-width, initial-scale=1.0">
<link rel="stylesheet" type="text/css" href="bootstrap.min.css"></link>
</head>
<body ng-app="coffeetech">
<div class="container" ng-controller="GithubCtrl">
<div>Subscriber count: {{ repo.subscribers_count }}</div>
<div>Network count: {{ repo.network_count }}</div>
</div>
<script
src="angular.js"></script>
<script src="github.js"></script>
<script src="coffeetech.js"></script>
</body>
</html>
```

我们可以修改 HTML，只显示仓库 JSON 数据中的几个重要信息。此处显示的是 `subscriber_count` 和 `network_count`。现在看到的页面整洁一些了（见图 9-2）。

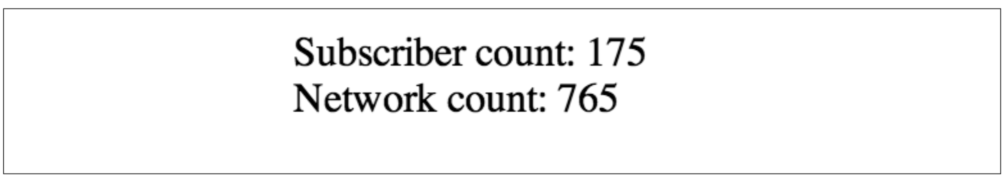


图 9-2：提取所关注的信息

我们使用 GitHub API 从 GitHub 托管的 Gollum 仓库中提取了订阅者和网络数量，然后把这些数据放到了单页应用中。

9.3.1　规划应用的数据结构

我们要构建的是一个咖啡店数据库，这个数据库存在 Git 中，但是 Git 及相关的工具（命令行工具或 GitHub）没有提供关系型数据库这种功能。因此，我们要规划一种结构，让存入仓库的数据易于搜索。

这个应用用于搜索咖啡店，而咖啡店基本都在大城市。我们可以把数据存在一些 JSON 文件中，各个文件使用城市名命名，这样数据就与城市关联起来了，在客户端可以使用地理定位功能或者让用户手动选择城市，然后获取相关的数据。

查看 GitHub 中 GitHub.js 的文档（https://github.com/michael/github）之后得知，从仓库中获取数据可以使用一些选项。我们将把数据存入以城市名命名的 JSON 文件中，然后

放入仓库中，再从那个仓库里获取数据。看起来我们要调用 `github.getRepo( username, reponame )` 获取仓库对象，然后再调用 `repo.contents( branch, path, callback )`。

现在应用有个骨架了，我们先停下来，确保构建出的应用易于重构和长期维护。这意味着，我们要在项目中添加测试。

9.3.2 让应用易于测试

测试能让我们写出更好的代码，因为我们能以局外人的身份思考代码该怎么使用，此外测试还能让外人（其他团队成员）更好地使用我们的代码。测试能促进“社交编程”。

接下来要使用的 JavaScript 测试工具是 Karma。这个工具简化了 JavaScript 单元测试的编写。在编写测试之前，先要安装这个工具。Karma 可以轻松地使用 `npm`（安装方法参见附录 B）来安装。

```
$ npm install karma -g
$ wget https://ajax.googleapis.com/ajax/libs/angularjs/1.2.7/angular-mocks.js
```

使用 angular-mocks.js 易于在测试中模拟 Angular 依赖。

然后，新建一个文件，将其命名为 karma.config.js，然后写入下述内容。

```
module.exports = function(config) {
  config.set({
    basePath: '',
    frameworks: ['jasmine'],
    files: [ ❶
        'angular.js',
        'fixtures-*.js',
        'angular-mocks.js',
        'firebase-mock.js',
        'github.js',
        '*.js'
    ],
    reporters: ['progress'],
    port: 9876,
    colors: true,
    logLevel: config.LOG_INFO,
    autoWatch: true,
    browsers: ['Chrome'], ❷
    captureTimeout: 60000,
    singleRun: false
  });
};
```

这与 Karma 的默认配置差不多。

❶ `files` 部分指定 JavaScript 实现脚本和测试脚本的加载顺序。可以看到，我们明确指定了前面添加的几个文件，然后使用通配符涵盖余下的文件。

❷ 还要注意，我们指定的测试浏览器是 Chrome（因此系统中要有）。使用这个浏览器最好，因为几乎所有桌面平台都支持。当然，如果你想让 Karma 在 Safari 或 Firefox 中测试，也可以指定这两个浏览器。Karma 会启动各个指定的浏览器实例，然后在其中创建测试用具，运行测试。

编写测试之前，要弄清楚想让代码做什么。

- 用户首次访问应用时，使用浏览器的地理定位功能确定用户所在的位置。
- 从仓库中获取包含各个城市经纬度的文件。
- 迭代城市列表，检查用户是否在这些城市方圆 25 英里的范围内。如果在的话，那就把当前城市设为第一匹配的城市。
- 确定所在城市后，从 GitHub 中加载对应的 JSON 数据。

具体而言，我们要断言能加载有两个城市的列表，其中有个城市名为 Portland，而且这个城市有三家咖啡店。

下面将使用 `ng-init` 指令，其作用是告诉 AngularJS 我们想在控制器加载完成后调用指定的函数。我们将把这个函数命名为 `init`。下面来测试这个函数。

首先要使用 Jasmine 测试框架为 AngularJS 应用的测试编写设置代码。Jasmine 是一个“行为驱动”JavaScript 库，提供了用于组织和创建预期行为的测试函数。Jasmine 框架为大多数常见的断言提供了“匹配器”（用于比较预期的结果和函数调用的返回值），如果没有合适的匹配器，那么还能自定义。此外，Jasmine 也支持为函数创建“侦件”，即截获函数调用，验证调用的方式是否符合预期。通过具体的测试最能体现 Jasmine 的优雅，下面我们就来编写测试。

```
describe( "GithubCtrl", function() {
    var scope = undefined; ❶
    var ctrl = undefined;
    var gh  = undefined;
    var repo = undefined;
    var geo = undefined;

    beforeEach( module( "coffeetech" ) ); ❷

    beforeEach( inject( function ($controller, $rootScope ) { ❸
            generateMockGeolocationSupport(); ❹
            generateMockRepositorySupport();
            scope = $rootScope.$new(); ❺
            ctrl = $controller( "GithubCtrl",
       { $scope: scope, Github: gh, Geo: geo } ); ❻
      } )
    );
...
```

❶ 在函数开头定义变量。如果不这么做，JavaScript 会在初次遇到变量时悄无声息地定义，

这会导致变量在设置代码和真正的测试中有差异。

❷ 在 beforeEach 函数中调用 module 方法加载 coffeetech 模块。beforeEach 函数中的代码会在各个测试运行之前执行。

❸ AngularJS 使用 inject 函数把 $controller 和 $rootScope 对象提供给 beforeEach 函数，用于设置测试。

❹ 调用两个函数，创建测试所需的模拟对象。稍后讨论这两个函数。

❺ 按照 AngularJS 的约定，所有功能和状态都存储在 scope 对象中。我们使用 AngularJS 提供的实用函数 $rootScope.$new() 新建一个 scope 对象，并把它的引用存储在一个变量中，以便测试代码中实现的真正功能。

❻ 传入模拟的对象和 scope 对象（由前面那两个函数调用创建），实例化一个控制器对象。这个控制器使用 scope 对象定义函数和数据，因为我们存储了 scope 对象的引用，所以可以调用相关的函数并检查数据，断言实现方式是正确的。

接下来，编写一个真正的测试。

```
        describe( "#init", function() { ❶
            it( "should initialize, grabbing current city", function() { ❷
                scope.init(); ❸
                expect( geo.getCurrentPosition ).toHaveBeenCalled(); ❹
                expect( gh.getRepo ).toHaveBeenCalled();
                expect( repo.read ).toHaveBeenCalled();
                expect( scope.cities.length ).toEqual( 2 ); ❺
                expect( scope.city.name ).toEqual( "portland" );
                expect( scope.shops.length ).toEqual( 3 );
            });
        });
    });
```

❶ describe 函数用于组织多个 it 函数中定义的测试。此处测试的是 init 函数，因此使用 #init 标识符是合理的。

❷ describe 块用于组织测试，而 it 块用于指定代码来运行测试。

❸ 控制器要先调用 init 函数，因此我们在测试中模拟这种行为，设置控制器的状态。

❹ 断言代码使用了我们在注入的对象上定义的不同接口：geo 对象上的 getCurrentPosition，以及仓库对象上的 read。

❺ 接下来，断言正确加载了数据。这里的测试验证有没有两个城市、有没有加载默认城市，以及默认城市的名字是不是等于字符串 "portland"。此外，测试还验证有没有为默认城市加载三个咖啡店。在实现代码中，这些数据从 JSON 文件中加载，不过我们关注的是接口和数据是否与预期相符。

如果你从未使用 Jasmine 为 JavaScript 代码写过测试，一开始可能觉得句法有点难懂，其实 Jasmine 的句法很优雅，为我们解决了大量问题。最为重要的一点是，Jasmine 提供了

spyOn 函数，我们可以使用它截获函数调用，然后断言函数被调用了。只要在测试中见到 toHaveBeenCalled()，就说明 spyOn 为我们提供了一个断言，用于确认函数被调用了。

现在要实现对上述测试至关重要的那两个模拟函数。把这两个函数放在 beforeEach(module("coffeetech")) 那行和 beforeEach(inject(...)) 函数之间，让 Karma 能调用它们。

```
...
beforeEach( module( "coffeetech" ) );

function generateMockGeolocationSupport( lat, lng ) { ❶
    response = ( lat && lng ) ?
        { coords: { lat: lat, lng: lng } } :
  { coords: CITIES[0] };
    geo = { getCurrentPosition: function( success, failure ) { ❷
        success( response );
    } };
    spyOn( geo, "getCurrentPosition" ).andCallThrough(); ❸
}

function generateMockRepositorySupport() { ❹
    repo = { read: function( branch, filename, cb ) { ❺
        cb( undefined,
      JSON.stringify( filename == "cities.json" ?
              CITIES : PORTLAND ) );
    } };
    spyOn( repo, "read" ).andCallThrough();

    gh = new Github({});
    spyOn( gh, "getRepo" ).andCallFake( function() { ❻
        return repo;
    } );
}

beforeEach( inject( function ($controller, $rootScope ) {
...
```

❶ 首先实现 generateMockLocation 函数。

❷ 为了模拟位置，我们要创建一个 geo 对象，而且要在这个对象上定义 getCurrentPosition 函数，这个函数在成功执行时调用 success 回调。这个过程与浏览器原生支持的地理定位功能完全一样，也定义了相同的函数。

❸ 然后，调用 spyOn 为 getCurrentPosition 函数创建侦件。这样才能在测试中断言 getCurrentPosition 函数被调用了。

❹ 接下来，实现 generateMockRepositorySupport 函数。

❺ 同样，我们要定义一个模拟对象，不过这一次要在对象上定义 read 方法。这个方法的作用与 JavaScript GitHub.js 库提供的 API 中的同名方法一样。与前面的模拟一样，我们调用 spyOn 为 read 方法创建侦件，这样才能在测试中断言 read 方法被调用了。可

是，repo 不是“顶层”对象，而是调用 getRepo 方法得到的对象。因此，要让 getRepo 调用返回这个模拟对象。

❻ 为 getRepo 方法创建侦件，返回模拟的仓库对象。调用 read 方法获取信息时使用的其实是这个对象。

写好测试之后，在命令行中运行测试组件，会看到测试失败。

```
$ karma start karma.conf.js
Chrome 32.0.1700 (Mac OS X 10.9.1) GithubCtrl #init should initialize,
          grabbing current city FAILED
  Error: [$injector:modulerr] Failed to instantiate module...:
  Error: [$injector:nomod] Module 'coffeetech' is not available!
    You either misspelled the module name or forgot to load it.
    If registering a module ensure that you specify the
    dependencies as the second argument.
...
```

接下来要提供一些测试固件（fixture）。

9.3.3　测试数据

我们要提供固件，即存储测试数据的文件。在保存其他代码的目录中新建 fixtures-cities.js 文件，写入下述内容。

```
var CITIES = [{
    name: "portland",
    latitude: 45,
    longitude: 45
}, {
    name: "seattle",
    latitude: 47.662613,
    longitude: -122.323837
}]
```

再新建 fixtures-portland.js 文件，写入下述内容。

```
var PORTLAND = [{
    "name": "Very Good Coffee Shop",
    "latitude": 45.52292,
    "longitude": -122.643074
}, {
    "name": "Very Bad Coffee Shop",
    "latitude": 45.522181,
    "longitude": -122.63709
}, {
    "name": "Mediocre Coffee Shop",
    "latitude": 45.520437,
    "longitude": -122.67846
}]
```

9.3.4　修改coffeetech.js文件

接下来，添加 coffeetech.js 文件。现在只关注设置代码和对 `init` 函数的改动。

```
var mod = angular.module( 'coffeetech', [] );

mod.factory( 'Github', function() {  // ❶
    return new Github({});
});

mod.factory( 'Geo', [ '$window', function( $window ) {  // ❷
    return $window.navigator.geolocation;
} ] );

mod.factory( 'Prompt', [ '$window', function( $window ) {
    return $window.prompt;
} ] );

mod.controller( 'GithubCtrl', [ '$scope', 'Github', 'Geo', 'Prompt',  // ❸
        function( $scope, ghs, Geo, Prompt ) {
    $scope.messages = []

    $scope.init = function() { // ❹
        $scope.getCurrentLocation( function( position ) {
            $scope.latitude = position.coords.latitude;
            $scope.longitude = position.coords.longitude;
            $scope.repo = ghs.getRepo( "xrd", "spa.coffeete.ch" );  // ❺
            $scope.repo.read( "gh-pages", "cities.json",
              function(err, data) {  // ❻
                $scope.cities = JSON.parse( data );  // ❼
                // 确认当前城市
                $scope.detectCurrentCity();  // ❽

                // 如果确定了所在城市,获取那个城市
                if( $scope.city ) {
                    $scope.retrieveCity();
                }

                $scope.$apply(); // ❾
            });
        });
...
```

❶ 提取 GitHub 库，放入一个 AngularJS 工厂方法中。这样便能把模拟的 GitHub 对象注入到测试中。如果把创建 GitHub 实例的代码放在控制器里，在测试中无法轻易模拟。

❷ 提取地理定位功能，放入一个 AngularJS 工厂方法中。与 GitHub 库一样，这样做可以向测试注入模拟的对象。

❸ 我们重新定义了控制器，注入所需的各个对象。我们提取出了 GitHub API 对象和 `Geo` 对象，让二者变成依赖，而这个句法能找到正确的对象，并把它们提供给控制器。此外，你可能注意到了，这一次创建控制器的句法稍微有点不同，我们使用的是

controller("CtrlName", ['dependency1', 'dependency2', function(dependency1, dependency2) {}])。使用这种写法，即使简化了 JavaScript 代码，代码也依然可用；而前面那种写法在简化后就失效了，因为 AngularJS 找不到依赖的名称。

❹ 把功能放入 `init` 函数中，这样就能在测试中调用了。

❺ 设置用户名，然后加载仓库。如果使用自己的仓库，那么要相应地做出修改，否则就继续使用这两个参数。

❻ 使用 `read` 方法读取仓库中文件的内容。注意，这里使用的是 `gh-pages` 分支，因为单页应用和所有数据都存在这个分支中。

❼ 获取到的数据只是一个字符串，要使用 `JSON.parse` 方法将其转换成 JavaScript 对象。

❽ 从仓库中获取数据之后，可以使用城市数组中的数据确认当前城市。

❾ 因为我们在 AngularJS 之外调用，而且在回调中返回，所以要像前面的示例那样调用 `scope.$apply()`。

接下来可以实现地理编码功能了。

9.4 添加地理编码功能

我们要定义几个函数，从 GitHub API 中获取城市数据，使用浏览器的地理定位功能找到用户所在的位置，使用用户的当前位置确定他们离哪个城市最近，计算距离，确定最近的城市后加载那座城市的数据，最后再定义一个函数查询用户的 GitHub 凭据，然后为数据添加评注。

首先，实现加载城市数据的函数。

```
$scope.retrieveCity = function() { ❶
    $scope.repo.read( "gh-pages", $scope.city.name + ".json",
      function(err, data) {
        $scope.shops = JSON.parse( data );
        $scope.$apply();
    });
}

$scope.loadCity = function( city ) { ❷
    $scope.repo.read( "gh-pages", city + ".json", function(err, data) {
        $scope.shops = JSON.parse( data );
        $scope.$apply();
    });
...
```

❶ `retrieveCity` 函数从仓库对象中读取咖啡店列表，方式与获取城市列表一样。把数据载入作用域中之后，调用 `$apply()`，通知 Angular。

❷ `loadCity` 加载城市名对应的城市数据。

接下来实现计算当前用户与各城市间距离的函数。

```
$scope.getCurrentLocation = function( cb ) { ❶
    if( undefined != Geo ) {
        Geo.getCurrentPosition( cb, $scope.geolocationError );
    } else {
        console.error('not supported');
    }
};

$scope.geolocationError = function( error ) { ❷
    console.log( "Inside failure" );
};

$scope.detectCurrentCity = function() { ❸
    // 计算两个位置之间的距离,以此确定
    // 离方圆25英里内的哪座城市最近
    for( var i = 0; i < $scope.cities.length; i++ ) {
        var dist = $scope.calculateDistance( $scope.latitude, ❹
                                             $scope.longitude,
                                             $scope.cities[i].latitude,
                                             $scope.cities[i].longitude );
        if( dist < 25 ) {
            $scope.city = $scope.cities[i];
            break;
        }
    }
}

toRad = function(Value) { ❺
    return Value * Math.PI / 180;
};

$scope.calculateDistance = function( latitude1,  ❻
                                     longitude1,
                                     latitude2,
                                     longitude2 ) {
    R = 6371;
    dLatitude = toRad(latitude2 - latitude1);
    dLongitude = toRad(longitude2 - longitude1);
    latitude1 = toRad(latitude1);
    latitude2 = toRad(latitude2);
    a = Math.sin(dLatitude / 2) * Math.sin(dLatitude / 2) +
        Math.sin(dLongitude / 2) * Math.sin(dLongitude / 2) *
        Math.cos(latitude1) * Math.cos(latitude2);
    c = 2 * Math.atan2(Math.sqrt(a), Math.sqrt(1 - a));
    d = R * c;
    return d;
...
```

❶ 定义代码中将调用的 `getCurrentLocation` 函数。我们在注入的 `Geo` 对象上调用 `getCurrentPosition` 函数来实现这个函数（在测试中要使用模拟的函数，因此在真正的代码中要在浏览器原生接口的基础上抽象）。

❷ 我们要为 getCurrentPosition 调用提供处理错误的回调，这里实现的就是这个回调，它把错误输出到控制台。

❸ 然后，定义 detectCurrentCity 函数，遍历城市列表，检查是否在某座城市中。

❹ 迭代城市列表，计算当前位置是否离哪座城市的距离小于 25 英里。各个城市都有经纬度数据。找到离得最近的城市后，把那座城市设为当前城市，然后退出循环。

❺ 为了计算距离，需要一个弧度转换函数。

❻ 最后，定义计算距离的函数。

一开始是不是觉得计算距离的函数有点难懂？我读过一篇文章之后才写出了这些代码。那篇文章介绍了如何使用 PostgreSQL 数据库中存储的过程（procedure）做地理编码，我把那些代码转成 JavaScript 了。如果不是地理编码专家，如何确定这些代码是正确的呢？好吧，我们编写几个测试检查一下。把下面几行代码添加到 coffeetech.spec.js 文件的末尾，放在最后一对结束括号 }); 内。

```
describe( "#calculateDistance", function() {
    it( "should find distance between two points", function() {
        expect( parseInt(
      scope.calculateDistance( 14.599512,
       120.98422,
       10.315699,
       123.885437 ) * 0.61371 ) ).
  toEqual( 354 );
    });
});
```

为了编写这个测试，我在 Google 中搜索了“distance between Manila”，Google 使用“Cebu”自动补全了我的搜索词条。Google 给出的距离是 338 英里。然后，我查出了这两座城市的经纬度，写出了上述测试。我预计测试会失败，因为根据坐标计算出的英里数多少会差一点。可是，测试表明两座城市距离为 571。奇怪，难道单位是千米而不是英里？确实，我忘了这种算法计算的是千米距离，而不是英里。因此，我们要把结果乘以 0.621371 才能得到英里数。最终的结果与 Google 给出的距离相差无几。

城市数据

我们要为应用提供一些可用的数据。把下述内容写入 cities.json 文件。

```
[
  {
    "longitude": -122.67620699999999,
    "latitude": 45.523452,
    "name": "portland"
  },
  {
    "longitude": -122.323837,
```

```
    "latitude": 47.662613,
    "name": "seattle"
  }
]
```

至此，我们实现了地理编码功能，还提供了示例数据。接下来要询问用户的登录凭据了。

9.5 添加登录功能

如果有人想派生 GitHub 中的仓库，那么必须登录 GitHub。因此，要让用户输入登录凭据。

```
...

$scope.annotate = function() {
    user = Prompt( "Enter your github username" )
    password = Prompt( "Enter your github password" )
    data = Prompt( "Enter data to add" );
};

...
```

现在，要像下面这样在 index.html 文件中显示这些新数据（省略了 HTML 文件中肯定都有的部分）。

```
<body ng-app="coffeetech">

<div class="container" ng-controller="GithubCtrl" ng-init="init()">

<h1>CoffeeTe.ch</h1>

<h3 ng-show="city">Current city: {{city.name}}</h3>

<div class="row">
<div class="col-md-6"><h4>Shop Name</h4> </div>
<div class="col-md-6"><h4>Lat/Lng</h4> </div>
</div>
<div class="row" ng-repeat="shop in shops"> ❶
<div class="col-md-6"> ❷
{{ shop.name }} ❸
</div>
<div class="col-md-6"> {{ shop.latitude }} / {{ shop.longitude }} </div>
</div>
</div>
```

❶ `ng-repeat` 是 AngularJS 提供的指令，用于迭代数组中的元素。这里，我们使用这个指令迭代 portland.json 文件中的元素，然后使用各次迭代从元素中获取的数据插入一段 HTML。

❷ 使用 Bootstrap 编写 HTML 结构很简单。`col-md-6` 类告诉 Bootstrap，创建一个占 12 栏布局（Bootstrap 默认的布局方式）一半宽度的分栏。我们使用这种方式并排放置了两

栏。如果在移动设备中访问，这两栏可能会纵向叠放。

❸ 使用 AngularJS 的双向数据绑定句法插入咖啡店的名字。

出错了吗

在浏览器中打开这个应用，不会看到我们所在城市中的咖啡店。这说明什么地方出错了，我们来调查调查。我推荐使用 Chrome 浏览器调试，不过也可以使用你喜欢的浏览器和开发者工具。如果使用的是 Chrome，那就在页面中的任意位置点击鼠标右键，在弹出菜单的底部选择“Inspect Element”（审查元素）（也可以按键盘上的 F12 快捷键，或者在 Windows 和 Linux 中按 Ctrl-Shift-I 键，在 Mac 中按 Cmd-Opt-I 键），打开开发者控制台。然后，选择控制台标签页。刷新浏览器窗口，你会在控制台中看到：

```
Uncaught TypeError: Cannot call method 'select' of undefined
```

点击右侧的 GitHub.js，会看到如图 9-3 所示的错误。

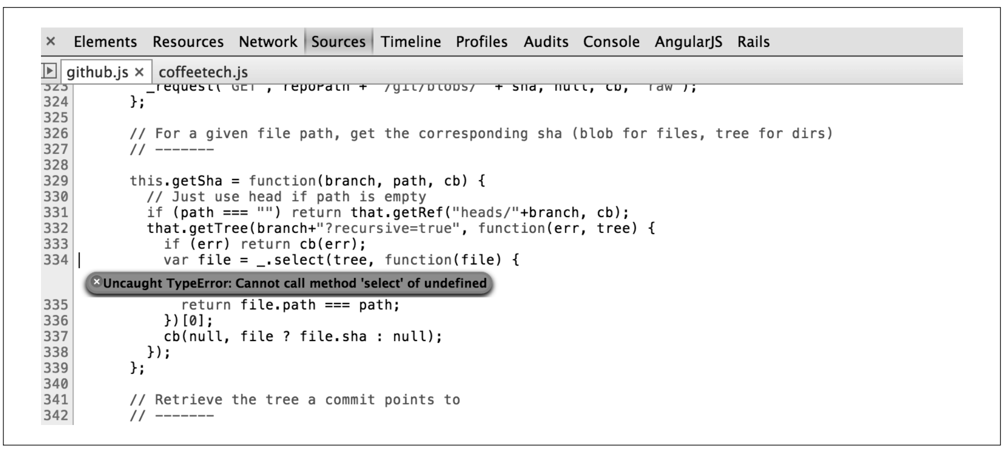


图 9-3：意料之外的错误

可以看到，这个错误是树对象上的 `select` 方法导致的。这个方法在下划线符号上定义。如果经常使用 JavaScript，你会意识到下划线变量出自 Underscore 库，而 `select` 方法的作用是获取数组中满足条件的第一个元素。根据 GitHub.js 库的实现，它会获取仓库的整个树对象，然后迭代其中的各个元素，选择匹配请求文件名的文件。这是需要考虑的重要性能因素。GitHub API 没有提供直接使用路径名请求内容的方式，我们要先获取文件列表，然后再使用 SHA 散列值请求文件。这个过程分两步，要调用 API 两次（可能会用很长时间）。

怎样修正未定义 `select` 方法的错误呢？是不是忘记引入 underscore.js 库了？根据 GitHub.js 文档，它需要使用 underscore.js 和 base64.js。我们忘记引入这两个库了。真是失误！为了引入这两个库，先在控制台中执行下述命令。

```
$ wget http://underscorejs.org/underscore-min.js
$ wget https://raw.github.com/dankogai/js-base64/master/base64.js
```

然后把这两个库添加到 index.html 文件中。最终，引入 JavaScript 文件的元素如下。

```
...

<script src="angular.js"></script>
<script src="underscore-min.js"></script>
<script src="base64.min.js"></script>
<script src="github.js"></script>
<script src="coffeetech.js"></script>
...
```

现在可以添加一些虚拟数据，规划用户提供的数据采用什么结构。

9.6 显示（即将）由用户提供的数据

目前，我们构建了存储城市和城市中咖啡店的数据库。可是，Google 地图或 Apple 地图已经提供了这些信息。不过，如果在此基础上添加额外的信息（例如关于咖啡店的奇怪信息），那么用户使用他们最爱的地图应用寻找咖啡店时可能会用得上。

首先添加一些关于咖啡店的虚拟数据。新建 portland.json 文件，写入下述内容。

```
[
   {
      "information" : [
         "offers gluten free desserts",
         "free wifi",
         "accepts dogs"
      ],
      "longitude" : -122.643074,
      "latitude" : 45.52292,
      "name" : "Very Good Coffee Shop"
   },
   {
      "latitude" : 45.522181,
      "name" : "Very Bad Coffee Shop",
      "longitude" : -122.63709
   },
   {
      "name" : "Mediocre Coffee Shop",
      "latitude" : 45.520437,
      "longitude" : -122.67846
   }
]
```

注意，我们在数据集中添加了一个名为 `information` 的数组。使用这个数组便于搜索。在 index.html 文件中添加搜索功能。

```
...

<div class="container" ng-controller="GithubCtrl" ng-init="init()">

<h1>CoffeeTe.ch</h1>

<input style="width: 20em;" ng-model="search"
       placeholder="Enter search parameters..."/> ❶

<h3 ng-show="city">Current city: {{city.name}}</h3>

<div class="row=">
<div class="col-md-6"><h4>Shop Name</h4> </div>
<div class="col-md-6"><h4>Lat/Lng</h4> </div>
</div>
<div class="row" ng-repeat="shop in shops | filter:search"> ❷
<div class="col-md-6">
{{ shop.name }}

<div ng-show="search"> ❸
<span ng-repeat="info in city.information">
<span class="label label-default">city.data</span>
</span>
</div>

</div>
<div class="col-md-6">
<a target="_map" ❹
   href="http://maps.google.com/?q={{shop.latitude}},{{shop.longitude}}">
   Open in map ({{shop.latitude}},{{shop.longitude}})
</a>
</div>
...
```

❶ 添加一个搜索框，绑定到作用域中的 search 模型上。

❷ 为数据添加一个过滤器，显示在 shops 数组中搜索得到的各个元素。

❸ 搜索时（定义模型变量 search）显示额外的信息。

❹ 使用经纬度信息指向 Google 地图中对应的页面。

现在，如果在搜索框中输入“gluten”，那么只会显示匹配词条的咖啡店，而且额外的信息以标注的形式显示在店名下面（见图 9-4）。



图 9-4：使用“gluten”词条过滤咖啡店

用户贡献的数据

现在应用可以正常运行了，下面我们要让用户添加信息，帮我们构建数据库。我们将在地图链接下方添加一个按钮，让用户为咖啡店添加额外的信息。

若想贡献信息，用户要派生仓库，改动之后向源仓库发起拉取请求。派生是指在 GitHub 账户中存一份源仓库的副本。这些步骤在这个 Web 应用中都能使用 GitHub.js 库实现。当然，如果有人想派生仓库到自己的账户中，那就必须登录，因此要让用户输入用户名和密码。如果你觉得在 Web 应用中让用户提供 GitHub 凭据不安全，先别急，稍后我们会使用安全的方式实现这个功能。

首先，要在 HTML 中添加 `annotate` 按钮。

```
<button ng-click="annotate(shop)">Add factoid</button>
```

然后，编写几个测试。新建 coffeetech.annotate.spec.js 文件，写入下述内容。

```
describe( "GithubCtrl", function() {

    var scope = undefined, gh = undefined,
      repo = undefined, prompter = undefined;

    function generateMockPrompt() {
        prompter = { prompt: function() { return "ABC" } }; ❶
        spyOn( prompter, "prompt" ).andCallThrough();

    }

    var PR_ID = 12345;
    function generateMockRepositorySupport() { ❷
        repo = {
            fork: function( cb ) {
                cb( false );
```

```
            },
            write: function( branch, filename, data, commit_msg, cb ) {
                cb( false );
            },
            createPullRequest: function( pull, cb ) {
                cb( false, PR_ID );
            },
            read: function( branch, filename, cb ) {
                cb( undefined,
                    JSON.stringify( filename == "cities.json" ?
                            CITIES : PORTLAND ) );
            }
        };
        spyOn( repo, "fork" ).andCallThrough();
        spyOn( repo, "write" ).andCallThrough();
        spyOn( repo, "createPullRequest" ).andCallThrough();
        spyOn( repo, "read" ).andCallThrough();

        gh = { getRepo: function() {} }; ❸
        spyOn( gh, "getRepo" ).andCallFake( function() {
            return repo;
        } );
        ghs = { create: function() { return gh; } };
    }

...
```

这与前面的测试类似，我们模拟了 GitHub.js 库中的几个对象。

❶ 模拟提示功能。我们将使用浏览器内置的提示机制提示用户输入用户名、密码和评注数据。

❷ 为模拟的 GitHub 对象添加三个新方法：`fork`、`write` 和 `createPullRequest`。然后验证有没有调用这三个方法。

❸ 调用 `getRepo` 函数时，我们想通过侦件确认的确调用了，不过我们还想返回为测试提供虚拟的仓库对象，因此要使用这个句法。

我们要在 `beforeEach` 函数中添加设置代码，为测试加载模拟的对象，并且创建控制器和作用域。

```
...

var $timeout;  // ❶
beforeEach( inject( function ($controller, $rootScope, $injector ) {
    generateMockRepositorySupport();  // ❷
    generateMockPrompt();
    $timeout = $injector.get( '$timeout' );  // ❸
    scope = $rootScope.$new();
    ctrl = $controller( "GithubCtrl",
       { $scope: scope,
         Github: ghs,
```

```
            '$timeout': $timeout,
            '$window': prompter } );
    } ) );
    ...
```

❶ 根据 GitHub.js 库中 fork 方法的文档，它要花点时间才能返回（GitHub 要花时间完成派生请求，具体多久不好确定），因此要在应用中设置超时时间，然后请求新仓库。使用 AngularJS 时，可以模拟一个可编程的超时界面，这样便于在测试中控制。

❷ 生成模拟的 GitHub 方法调用和侦件，然后生成模拟的提示功能。

❸ 前面说过，我们要获取 $timeout 变量，使用这个方法可以从 injector 对象上获取 AngularJS 为测试提供的模拟超时时间。

接下来，为 annotate 函数编写测试。

```
...
describe( "#annotate", function() { ❶
    it( "should annotate a shop", function() {
        scope.city = PORTLAND
        var shop = { name: "A coffeeshop" }
        scope.annotate( shop ); ❷
        expect( scope.shopToAnnotate ).toBeTruthy(); ❸
        expect( prompter.prompt.calls.length ).toEqual( 3 );
        expect( scope.username ).not.toBeFalsy();
        expect( scope.annotation ).not.toBeFalsy();

        expect( repo.fork ).toHaveBeenCalled(); ❹
        expect( scope.waiting.state ).toEqual( "forking" ); ❺
        $timeout.flush(); ❻

        expect( scope.forkedRepo ).toBeTruthy(); ❼
        expect( repo.read ).toHaveBeenCalled();
        expect( repo.write ).toHaveBeenCalled();
        expect( repo.createPullRequest ).toHaveBeenCalled();
        expect( scope.waiting.state ).toEqual( "annotated" );
        $timeout.flush(); ❽

        expect( scope.waiting ).toBeFalsy();
    });

});
...
```

❶ 新建一个 describe 块，将其命名为 #annotate，用于组织相关的测试。然后，实现一个 it 函数，把描述信息设为“annotate a shop”。我们只会编写这一个测试。

❷ 设置好前置条件之后，scope 对象应该有个选中的城市，然后创建一个咖啡店，再调用 annotate 方法为咖啡店添加评注。

❸ 调用 annotate 方法之后，代码应该请求 GitHub API 的登录凭据，然后让用户提供对咖啡店的评注。在浏览器中，这个过程会出现三个提示。测试模拟了 prompt 对象，因此

应该调用模拟的 prompt 对象三次。此外，我们还验证了 scope 对象应该具有的几个状态，例如供后面使用的用户名和评注。

❹ 然后，GitHub API 应该执行第一个调用：GitHub.js 应该请求派生仓库。

❺ 随后应该进入等待状态。UI 会使用 scope.waiting.state 通知用户，我们正在等待。

❻ 模拟派生完成的超时时间结束后，我们会看到代码在作用域中存储了派生仓库得到的结果。

❼ 接下来，验证 GitHub API 调用了添加评注的方法。

❽ 再次调用 flush() 让超时时间结束。最后，全部结束之后，应该告诉用户不再处于等待状态了。

如果后台还运行着 Karma，会看到测试失败，输出下述消息：

```
Chrome 32.0.1700 (Mac OS X 10.9.1) GithubCtrl #annotate should
annotate a shop FAILED
        TypeError: Object #<Scope> has no method 'annotate'
            at null.<anonymous> (/.../coffeetech.spec.js:80:19)
```

接下来，我们要在 coffeetech.js 文件中实现这个功能。在文件的末尾，结束花括号之前添加下面几行代码。annotate 函数其实做了两件事：派生仓库到用户名下；等派生结束后，使用 GitHub API 在派生的仓库中添加评注信息。

```
...
$scope.annotate = function( shop ) { ❶
    $scope.shopToAnnotate = shop;
    $scope.username = $window.prompt( "Enter your github username (not email!)" )
    pass = $window.prompt( "Enter your github password" )
    $scope.annotation = $window.prompt( "Enter data to add" ); ❷
    gh = ghs.create( $scope.username, pass ); ❸
    toFork = gh.getRepo( "xrd", "spa.coffeete.ch" ); ❹
    toFork.fork( function( err ) {
        if( !err ) { ❺
            $scope.notifyWaiting( "forking",
              "Forking in progress on GitHub, please wait" ); ❻
            $timeout( $scope.annotateAfterForkCompletes, 10000 ); ❼
            $scope.$apply();
        }
    } );

};
  ...
```

❶ 首先，定义 annotate 函数。参照测试，这个函数应该接收一个 shop 对象作为参数，评注添加到这个对象上。

❷ 提示用户三次，分别让用户输入 GitHub 的用户名和密码，以及评注的文本。如果你觉得这样做很不妥，别担心，稍后会修改。

❸ 使用用户输入的用户名和密码创建一个 GitHub 对象。对拼写不对或错误凭据的处理留作练习交给读者完成。

❹ 使用 GitHub.js 库提供的 `getRepo` 函数可以创建仓库对象（在本地创建对现有仓库的引用）。有了仓库对象之后，可以调用 `fork` 方法派生仓库。

❺ 即便没有错误，也要意识到派生操作所需的时间不可测。因此，我们等待 10 秒钟再检查以确保派生完成。因为这个操作在浏览器中执行，没办法收到通知，所以必须轮询 GitHub，检查派生有没有完成。在实际使用时，如果发现没有结束，或许要再次检查，以防 GitHub 正在操作中。

❻ 使用 `"forking"` 键注册一个消息，我们可以在 HTML 模板中显示这个消息，告诉用户正在复刻中。这个函数稍后定义，它的基本作用是存储这个键和要显示的字符串，当消息为空时就去掉通知。

❼ 最后，调用 `annotateAfterForkCompletes` 方法，在整个过程完全结束后向派生的仓库添加数据。

下面编写派生完成后向仓库添加评注的代码：

```
...

$scope.annotateAfterForkCompletes = function() { ❶
    $scope.forkedRepo = gh.getRepo( $scope.username, "spa.coffeete.ch" );
    $scope.forkedRepo.read( "gh-pages", "cities.json", function(err, data) {
        if( err ) {
            $timeout( $scope.annotateAfterForkCompletes, 10000 );
        }
        else {
            $scope.notifyWaiting( "annotating",
              "Annotating data on GitHub" ); ❷
            // 把新数据写入仓库
            $scope.appendQuirkToShop();

            var newData = JSON.stringify( $scope.shops, stripHashKey, 2 ); ❸
            $scope.forkedRepo.write('gh-pages', $scope.city.name + '.json', ❹
                                    newData,
                                    'Added my quirky information',
                                    function(err) {
                if( !err ) {
                    // 发起评注数据的拉取请求
                    var pull = { ❺
                        title: "Adding quirky information to " +
                          $scope.shopToAnnotate.name,
                        body: "Created by :" + $scope.username,
                        base: "gh-pages",
                        head: $scope.username + ":" + "gh-pages"
                    };
                    target = gh.getRepo( "xrd", "spa.coffeete.ch" ); ❻
                    target.createPullRequest( pull,
                      function( err, pullRequest ) { ❼
```

```
                        if( !err ) {
                            $scope.notifyWaiting( "annotated",
                              "Successfully sent annotation request" ); ❽
                            $timeout(
                              function() {
                               $scope.notifyWaiting( undefined )
                              }, 5000 );
                            $scope.$apply(); ❾
                        }
                    } );
                }
                $scope.$apply();
            });
        }
        $scope.$apply();
    } );

...
```

❶ 确认派生结束后，我们需要获取新派生的仓库。使用用户登录时提供的用户名构建仓库对象，然后从仓库中读取 cities.json 文件。如果能成功读取这个文件（`err` 对象的值不为真），就表明可以开始编辑数据了。

❷ 通知 UI 我们正在发表评注，告诉用户在处理评注请求的过程中要等待。

❸ `JSON.stringify` 把对咖啡店的评注对象转换成 JSON 对象。即便你以前用过 `JSON.stringify`，可能也不知道它还有两个参数（除想序列化的对象之外）。第二个参数的作用是序列化时过滤对象，忽略指定的元素。第三个参数的作用是指定是否以及如何缩进得到的 JSON。我们把第二参数指定为 `stripHashKey` 函数，删掉 Angular 跟踪数据的 `$$hashKey`，把第三个参数设为缩进量。缩进后拉取请求更易阅读，因为 `JSON.stringify` 默认把对象序列化成一个长长的 JSON 字符串不易于计算差异，而缩进后可以逐行计算差异。

❹ 然后，使用 `write` 函数把数据写入派生的仓库中。如果写入成功，最后一个参数，也就是回调函数中的错误值会是未定义的。

❺ 如果错误是未定义的，我们可以向源仓库发起拉取请求。为此，我们要创建一个拉取请求，将其提供给 GitHub.js 库中发起拉取请求的方法。

❻ 获取拉取请求目标（即源仓库）的引用。

❼ 然后，向目标发起拉取请求。这个函数的第一个参数是前面创建的拉取请求对象，第二个参数是一个回调。如果请求失败，回调会收到一个错误代码；否则会收到一个拉取请求对象。

❽ 请求成功后，我们可以通知 UI，告诉它评注过程结束了，等 5000 毫秒（即 5 秒）后再把通知消息删掉。

❾ 前面说过，只要在第三方库（如 GitHub.js）的回调中，就要使用 `$apply()` 通知 Angular，告诉它作用域对象变了。

我们要实现三个辅助方法：

```
...

$scope.appendQuirkToShop = function() { ❶
    if( undefined == $scope.shopToAnnotate.information ) {
        $scope.shopToAnnotate.information = [];
    }
    $scope.shopToAnnotate.information.push( $scope.annotation );
};

function stripHashKey( key, value ) { ❷
    if( key == "$$hashKey" ) {
        return undefined;
    }
    return value;
}

$scope.notifyWaiting = function( state, msg ) { ❸
    if( state ) {
        $scope.waiting = {};
        $scope.waiting.state = state;
        $scope.waiting.msg = msg;
    }
    else {
        $scope.waiting = undefined;
    }
}
...
```

❶ 在 appendQuirkToShop 函数中，我们判断评注数组有没有定义；如果没有定义，那么创建一个空数组，然后把评注添加到里面。这么做是为了保证向未定义的数组中添加对象时应用不会崩溃。

❷ 定义在 JSON.stringify 函数中使用的转换函数。使用 ng-repeat 指令时，AngularJS 会向对象中添加一个跟踪属性（$$hashKey），这个函数的作用是把它从拉取请求数据中过滤掉。

❸ notifyWaiting 的作用（显然）是通知用户。创建一个 waiting 对象，然后更新状态（应用通过这个状态判断要不要显示消息），并指定一个消息。如果提供空消息，那就意味着销毁对象，也就是从 UI 中删除消息。

接下来要修改 HTML，把状态消息开放给 UI 使用。

```
...
<input class="ctinput" ng-model="search"
       placeholder="Enter search parameters..."/>

<h3 ng-show="city">Current city: {{city.name}}</h3>

<div ng-show="waiting">
```

```
    {{waiting.msg}}
  </div>
  ...
```

9.7 接受拉取请求

有人为咖啡店添加评注后，源仓库的属主会收到 GitHub 发送的拉取请求通知（见图 9-5）。

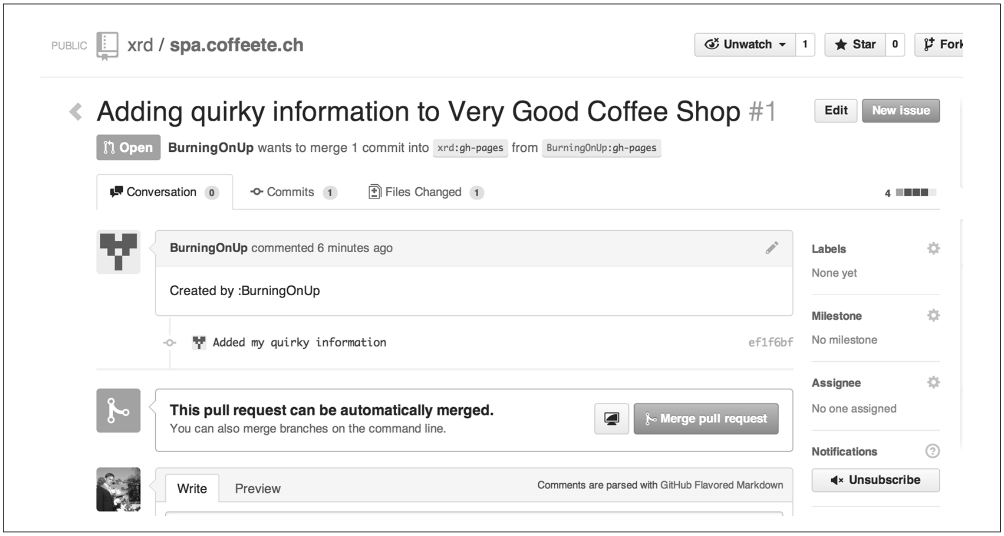


图 9-5：通过拉取请求添加信息

我们可以通过 GitHub 集成的在线差异工具审查改动（见图 9-6）。

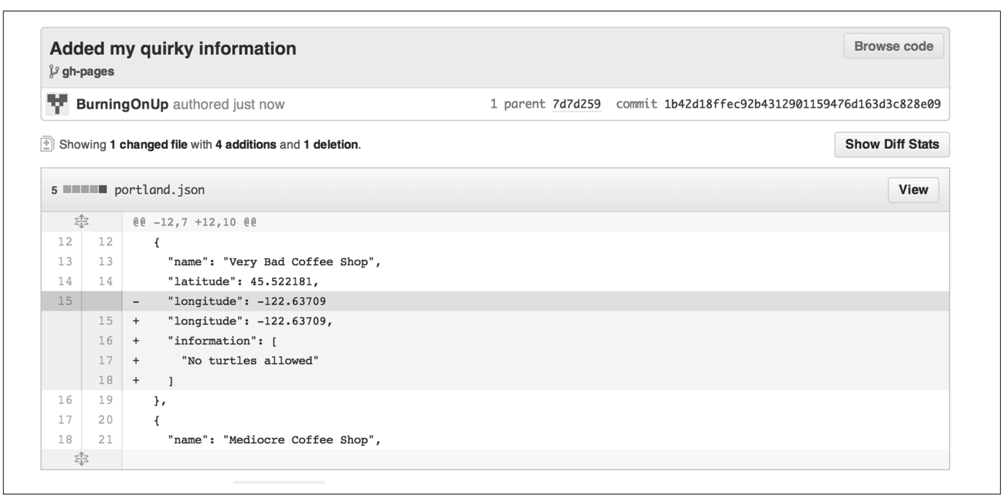


图 9-6：在 GitHub 中审查评注拉取请求的差异

我们可以清楚地看到贡献者所做的改动：用户添加了一条评注，说“禁止携带乌龟”。既然这样，那么下次与 Morla 约会时可能要考虑换个地方了。差异中以绿色显示的信息能吸引阅读者的目光，只要不把 `JSON.stringify` 的第三个参数设为未定义，就能获得这个效果。可惜，第一行的差异只是多个逗号，不过整体依然十分便于阅读。

9.8　实现安全的登录方式

如果我在网上遇到这个应用，那我绝对不会使用它提交数据，因为它要我输入 GitHub 的用户名和密码。这么做是想让我信任应用的作者。我所说的“信任”是指确信应用的作者不会恶意使用我的凭据做不法的事，还要确信应用的作者没有埋下漏洞，不会让攻击者入侵身份验证过程，把我的凭据偷走。我的很多在线服务都使用 GitHub 作为身份验证方式，因此我绝不会轻易把凭据提供给一个 Web 应用。

幸好，询问密码还有其他方式：OAuth。

使用 OAuth 的话，用户直接在 GitHub 中输入凭据。如果用户开启了双重身份验证，GitHub 依然能验证用户的身份（不过我们这个简单的实现无法处理这种身份验证过程）。输入凭据后，GitHub 会判断我们是不是声称的那个人，然后返回应用，赋予请求的权限。

> 使用 OAuth 有很多好处。GitHub 会为应用提供 OAuth 令牌，令牌中封装着有权在 GitHub 中执行的操作、是不是只读权限，以及能不能读写数据。因此，请求权限的服务可以只要求修改 GitHub 中的部分数据。这样用户会更相信应用，因为他们知道应用不能触碰他们在 GitHub 中保存的私有数据。比如说，我们可以只请求访问 Gist 的权限，而不请求访问仓库的权限。而且 OAuth 令牌有一大优点：可以吊销。因此，当执行特定的操作时，我们可以销毁令牌，撤销访问权限。如果使用用户名和密码，只有更换密码才能撤销访问权限；这意味着所有使用那个密码的地方（密码管理工具，以及使用用户名和密码登录的其他应用）都要修改。使用 OAuth 时，随时可以吊销令牌（在 GitHub 中操作很简单），而且其他服务不会受到影响。

下面我们来修改应用，使用 OAuth。

9.8.1　身份验证需要服务器

截至目前，我们把所有文件都发布到 GitHub 中了，也就是都交给 GitHub 托管了。可惜的是，身份验证组件不能托管在 GitHub 中，因为我们要通过 GitHub 以安全的方式验证用户的身份，获取 OAuth 令牌。目前，没有完美的客户端方案（只能使用运行在浏览器中的静态 HTML 和 JavaScript）能做到这一点。其他身份验证提供商，例如 Facebook，在 SDK 中提供了纯 JavaScript 登录功能，但是 GitHub 基于安全考虑，尚未发布纯客户端身

份验证方案。

因此，为了验证身份，我们要使用服务器。最简单的做法是运行一个小型身份验证服务器，把身份验证过程交给它处理，处理完毕后，再返回 GitHub 中托管的应用。我在本章对应的仓库中提供了相关代码（使用服务器端 JavaScript，即 NodeJS 编写）。可是，即便是简单的身份验证系统，这么做也有点小题大做。如果能把身份认证过程交给第三方处理，系统的代码量和复杂程度都将大大减少。

9.8.2　使用Firebase处理身份验证过程

我们不会自己搭建服务器，让它与 GitHub API 通信以及验证身份，而是把这个过程交给 Firebase 来做。Firebase 是实时通信工具，能很好地与我们选择的 AngularJS 集成。目前最简单也是最安全的方案是，使用 Firebase 提供的 AngularJS 绑定（“AngularFire”），以及集成好的 GitHub 身份验证组件（“Simple Login”）。这两个库能为我们解决身份验证问题，而且所有代码还都托管在 GitHub 中。把身份验证过程交给 Firebase 处理很简单，我们只需修改托管在 GitHub 中的应用，把凭据和 GitHub OAuth 作用域提供给 Firebase，这样用户管理功能就交给 Firebase 去做了。

首先创建一个新 GitHub 应用。在 GitHub.com 的右上角点击“Settings”链接，在打开的页面中点击靠近底部的“OAuth applications”链接。在右栏中点击“Developer applications”标签页，然后点击“Register a new application”按钮。“Authorization callback URL”中要填写 https://auth.firebase.com/auth/github/callback。然后，点击“Register application”按钮，保存应用，如图 9-7 所示。[1]

接下来，注册一个 Firebase 账户。然后，在 Firebase 中新建一个应用，命名为“CoffeeTech”。应用的 URL 必须独一无二，你可以使用“coffeetech-<USERNAME>”，记得把 USERNAME 换成你的 GitHub 用户名。创建应用后，点击“View Firebase”按钮。此时你会看到设置界面，点击“Simple Login”，然后点击“GitHub”，如图 9-8 所示。

注 1：其中链接文本和按钮文本已根据 GitHub 的新版 UI 做了更新。——译者注

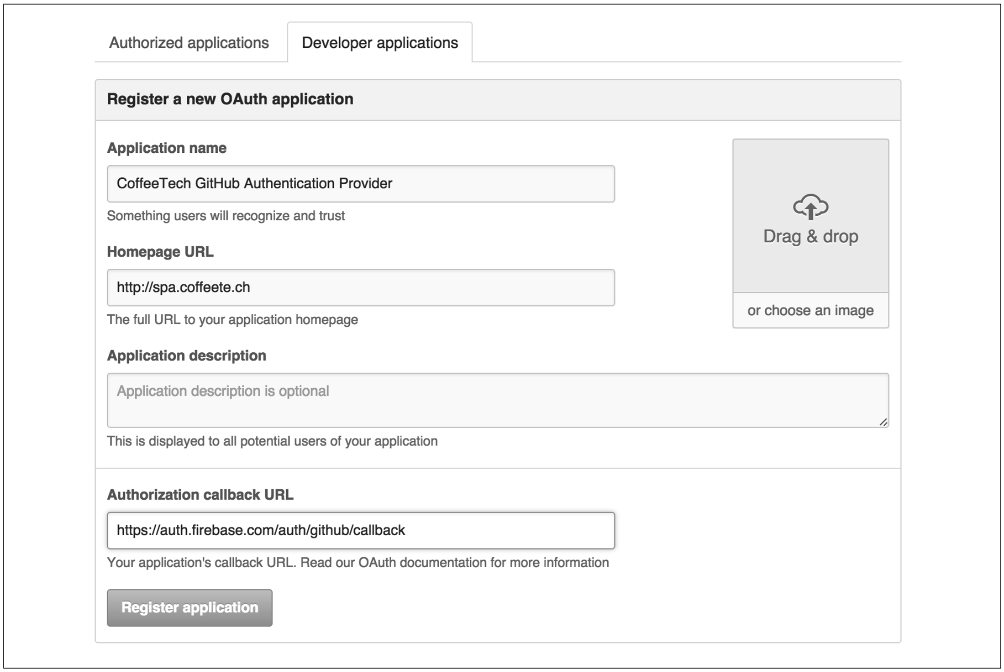


图 9-7：为了使用 OAuth，新建一个 GitHub 应用

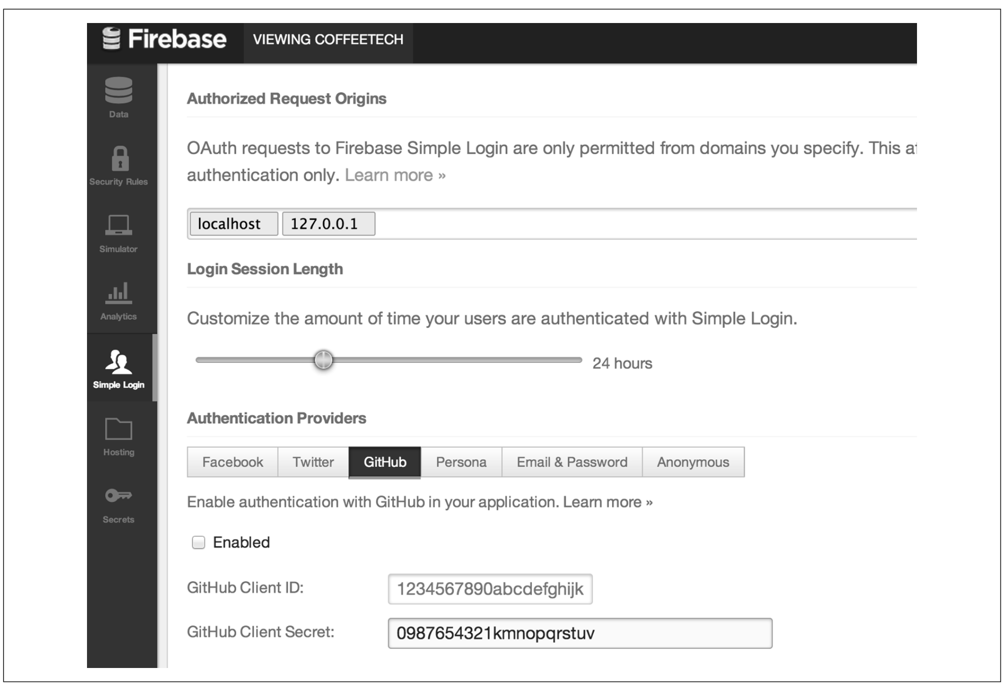


图 9-8：把登录过程交给 Firebase 处理

最后，复制 GitHub 客户端 ID 和密令，粘贴到 Firebase Simple Login 界面中设置 GitHub 提供商的部分。记得勾选“Enabled”，启用这个提供商。

至此，我们在 GitHub 中创建了一个登录应用，做了相关配置，让它使用 Firebase 服务，还配置了 Firebase，让它使用那个 GitHub 应用。我们希望所有功能，尤其是外部服务，都被测试覆盖。那么，接下来编写覆盖这个功能的测试。

9.8.3 测试Firebase

由于我们从 CDN 中加载 Firebase，首先要使用简单的 shim 来模拟 `Firebase` 构造方法。把下述代码写入 firebase-mock.js 文件：

```
var Firebase = function (url) {
}

angular.module( 'firebase', [] );
```

为了测试代码，我们要对 coffeetech-annotate.spec.js 文件做下述改动：

```
beforeEach( module( "coffeetech" ) );

var mockFirebase = mockSimpleLogin = undefined;
function generateMockFirebaseSupport() { ❶
    mockFirebase = function() {};
    mockSimpleLogin = function() {
        return {
            '$login': function() {
                return { then: function( cb ) {
                    cb( { name: "someUser",
                          accessToken: "abcdefghi" } );
                } };
            }
        }
    };
}

var $timeout;
beforeEach( inject( function ($controller, $rootScope, $injector ) {
    generateMockRepositorySupport();
    generateMockPrompt();
    generateMockFirebaseSupport(); ❷
    $timeout = $injector.get( '$timeout' );
    scope = $rootScope.$new();
    ctrl = $controller( "GithubCtrl",
       { $scope: scope,
         Github: ghs,
         '$timeout': $timeout,
         '$window': prompter,
         '$firebase': mockFirebase,
         '$firebaseSimpleLogin': mockSimpleLogin } ); ❸
```

```
} ) );

describe( "#annotate", function() {
    it( "should annotate a shop", function() {
        scope.auth = mockSimpleLogin( mockFirebase() ); ❹
        scope.city = PORTLAND
        var shop = { name: "A coffeeshop" }
        scope.annotate( shop );
        expect( prompter.prompt.calls.length ).toEqual( 1 ); ❺
        expect( scope.shopToAnnotate ).toBeTruthy();
        expect( scope.username ).not.toBeFalsy();
        expect( scope.annotation ).not.toBeFalsy();

        expect( repo.fork ).toHaveBeenCalled();
        expect( scope.waiting.state ).toEqual( "forking" );
        $timeout.flush();

        expect( scope.forkedRepo ).toBeTruthy();
        expect( repo.read ).toHaveBeenCalled();
        expect( repo.write ).toHaveBeenCalled();
        expect( repo.createPullRequest ).toHaveBeenCalled();
        expect( scope.waiting.state ).toEqual( "annotated" );
        $timeout.flush();

        expect( scope.waiting ).toBeFalsy();
```

❶ 添加 `generateMockFirebaseSupport()` 函数，用于创建模拟的 `Firebase` 对象和简单登录对象。

❷ 调用这个函数，创建驭件。

❸ 在测试中，我们使用 `$controller` 实例化方法注入这些驭件，而不让 AngularJS 注入真正的对象。现在，需要向控制器注入对象的测试都要这样修改。

❹ 修改 `#annotate` 测试，创建 `auth` 对象（通常在初始化时创建）。

❺ 只需为评注数据提示一次（不再需要提示输入用户名和密码了）。

9.8.4 实现Firebase登录功能

现在，为我们的 AngularJS 应用添加 Firebase 支持。在加载 AngularJS 的元素后面引入 Firebase 的支持库。

```
<script src="angular.js"></script>
<script src='https://cdn.firebase.com/v0/firebase.js'></script>
<script
  src='https://cdn.firebase.com/libs/angularfire/0.6.0/angularfire.min.js'>
</script>
<script
  src='https://cdn.firebase.com/js/simple-login/1.2.5/firebase-simple-login.js'>
</script>
```

我们要对 coffeetech.js 文件做几处改动。首先，把 Firebase 导入 AngularJS 模块。此外，之前要把用户名和密码作为参数传给 GitHub 服务，但是现在我们要使用稍微不同的方法签名，传入 OAuth 令牌。

```
var mod = angular.module( 'coffeetech', [ 'firebase' ] );

mod.factory( 'Github', function() {
    return {
        create: function(token) {
            return new Github( { token: token, auth: 'oauth' } );
        }
    };
});
```

实例化控制器时，要注入 `Firebase` 和 `FirebaseSimpleLogin`，然后在 `init` 方法中初始化。

```
mod.controller( 'GithubCtrl', [ '$scope', 'Github', 'Geo', '$window', '$timeout',
    '$firebase', '$firebaseSimpleLogin',
    function( $scope, ghs, Geo, $window, $timeout,
        $firebase, $firebaseSimpleLogin ) {

    $scope.init = function() {

        var ref = new Firebase( 'https://coffeetech.firebaseio.com' );
        $scope.auth = $firebaseSimpleLogin( ref );

        $scope.getCurrentLocation( function( position ) {
            $scope.latitude = position.coords.latitude;
```

然后，发表评注时，要提供 Firebase 返回的 `auth` 令牌。除此之外，还要对流程做些小改动。

```
$scope.annotate = function( shop ) {
    $scope.shopToAnnotate = shop;

    $scope.auth.$login( 'github', { scope: 'repo' } ).then(
      function( user ) { ❶

        $scope.me = user;
        $scope.username = user.name;

        $scope.annotation = $window.prompt( "Enter data to add" ); ❷

        if( $scope.annotation ) {
            gh = ghs.create( $scope.me.accessToken ); ❸
            toFork = gh.getRepo( "xrd", "spa.coffeete.ch" );
            toFork.fork( function( err ) {
```

❶ 在 Firebase SimpleLogin 服务创建的 `auth` 对象上调用 `$login` 方法。这个方法返回一个 promise 对象，这个接口有个 `then()` 方法，在登录成功时调用。`then()` 方法调用指定的回调函数，提供用户对象。

❷ 我们还要提示用户输入一个信息——评注数据。这个信息可以使用其他方式获取，例如使用 HTML5 模式对话框，不过现在这么做已经能达到目的。毕竟，现在只提示一次，而不用提示三次。

❸ 准备好派生时，要使用令牌创建用户对象。

改好之后，可以点击“Add factoid”按钮，此时会打开如图 9-9 所示的界面，这表明我们将使用 GitHub 登录（通过 Firebase SimpleLogin 服务）。

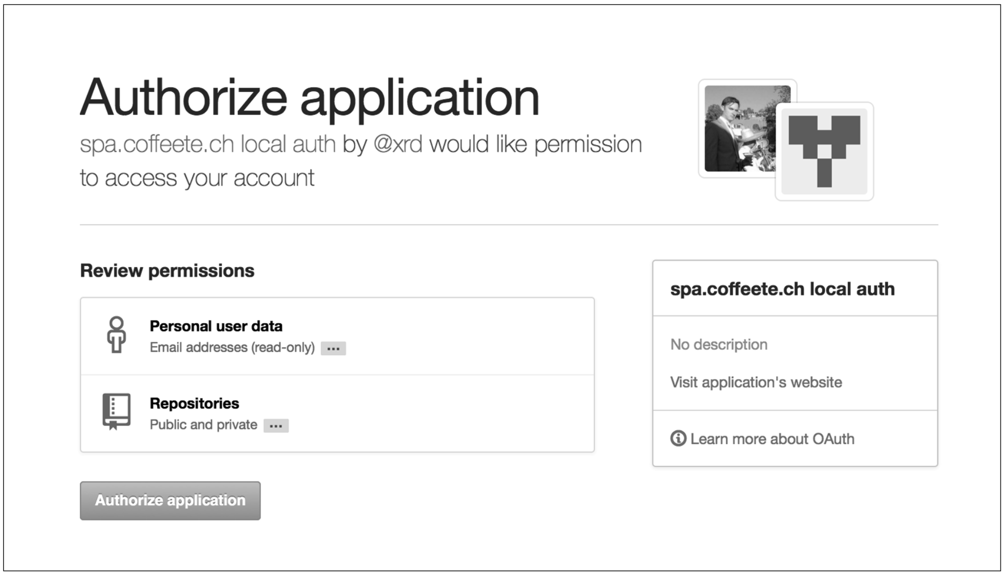


图 9-9：使用 Firebase 通过 GitHub 登录的最后一步

验证身份之后的执行流程与之前（使用用户名和密码）一样。我们可以优化一下，在调用 `$login()` 方法之前检查用户是否已经登录，不过这里先不这么做，也就是说每次点击那个按钮都会立刻跳出登录界面。

用户登录后，会重定向到应用，我们可以通知他们，告诉他们所做的贡献已通过拉取请求提交了。用户的贡献与他们的 GitHub 账号是关联的，拉取请求被接受后会收到标准的通知，因此此处不用再实现通知功能。

9.9　小结

本章使用 JavaScript 构建了一个不用服务器的应用，这个应用为用户提供可搜索的咖啡店数据库，而且可以使用 Pull Request API 以安全的方式接受贡献。我们可以完全忽略数据系统的管理功能，而将其交给 GitHub 处理。我们的单页应用只关注一件事：开发强大且实用的应用。

附录A

GitHub企业版

大多数人把 GitHub（公司）与 GitHub.com（网站）划等号，但是二者并非同一个事物。

现在，开源和闭源软件都喜欢托管在 GitHub 网站中，但这不是 GitHub 公司的唯一产品。除此之外，GitHub 公司的另一个主力产品是 GitHub 企业版。这个版本可以部署到企业的防火墙之后，相当于企业专用的 GitHub.com。

在用户看来，这两个产品非常相似；其实，二者之间有明显的差别。有时很难想象企业版能解决什么困境，但是要知道，它是针对大型团队的。

A.1　安装

使用 GitHub 企业版可不是注册一个账户那么简单，我们自己要负责全部基础设施和维护工作，包括安装、升级、系统维护、让设备始终运行，等等。不过，如果你的公司决定使用企业版，说明公司已经雇用了熟知这些操作的专业人士。

GitHub 团队努力做得更好，让维护人员更轻松。企业版自带不同格式的虚拟机，很可能有一种符合贵公司基础设施的要求。虚拟机运行起来之后，大多数配置都可以在 Web 界面中设置，不过有些棘手的配置，如网络配置和端口转发，外行很难做到正确设置。

A.2　管理

企业版的运行环境由我们自己负责，这可不像 GitHub.com 用户那么轻松，我们要关注很多问题。GitHub 企业版提供了管理界面，用于处理这些问题。这个界面是 GitHub.com 没有的，在这里可以做很多事情，例如管理系统资源、报告、搜索，等等。

此外，GitHub.com 有自己的一套用户系统，而企业版能与公司现有的身份认证系统集成。

这样，公司的 IT 部门可以在一处管理所有用户身份，雇用新员工时无需在多处创建账户。一开始切换系统时可能要为几千个人创建新账号，不过既然能集成现有系统，整个过程就会变得简单。GitHub 企业版支持多种身份验证系统，包括 LDAP、SAML，以及传统的电子邮件和密码。

A.3 端点

GitHub API 在企业版中都可以使用，只不过请求的地址不是 https://api.github.com/，而是 https://<hostname>/api/v3。可以想象，有些用户同时拥有企业版和 GitHub.com 账户，而且很多应用都开始支持这种情况。

A.4 单独的主机名与挂载点

GitHub.com 与企业版的主要区别通常是主机名不同。GitHub.com 为不同的内容提供了不同的主机名，下面是不完整的列表。

- github.io
  存贮用户和项目的 Jekyll 博客

- gist.github.com
  存贮 Gist

- raw.githubusercontent.com
  存贮原始页面（未经处理的文件）

基于多方面的原因，GitHub 企业版没有提供这样的映射关系。企业版安装可能是下面这样的。

- github.bigdevcorp.example.com/pages/xrd/somerepo
  存贮 gh-pages 网站

- github.bigdevcorp.example.com/gists
  存贮 Gist

可以看出，企业版通常把子域名映射到子目录上，而不是单独的主机名。这样简化了企业版的安装。不过，这意味着有些工具要重新配置。

对命令行工具 Gist（https://github.com/defunkt/gist）来说，我们要导出一个环境变量，指定 Gist 的 URL：

```
$ export GITHUB_URL=http://github.bigdevcorp.example.com/
```

对命令行工具 Hub（https://github.com/github/hub）来说，我们要导出另一个环境变量——GITHUB_HOST：

```
$ GITHUB_HOST=github.bigdevcorp.example.com hub clone myproject
```

A.5 命令行客户端工具：cURL

第 1 章说明了如何使用 cURL 向 GitHub.com 网站的 API 发送请求。如果想使用 cURL 请求企业版网站，方式有点不同：

```
$ curl -i https://github.bigdevcorp.example.com/api/v3/search/repositories?q=@ben
```

A.6 使用客户端库的请求示例

大多数客户端库都支持配置使用不同的端点，而使用 GitHub 企业版时必须这么做。

本书演示如何使用 5 门语言连接 GitHub：Ruby、Java、JavaScript、Python 和 C#。下面是各门语言的示例。参照这些代码片段，你可以对书中的任何示例进行改造，让它适应 GitHub 企业版。

A.6.1 Ruby

对 Ruby 客户端 Octokit 来说，要使用类似下面的代码：

```
github = Github.new
           basic_auth: 'login:password',
           endpoint: 'https://github.bigdevcorp.example.com/api/v3/'
puts github.repos.list
```

A.6.2 Java

对 Java 库 EGit 来说，要使用下述方式指定企业版端点：

```
GitHubClient client = new GitHubClient("github.bigcorpdev.example.com");
UserService us = new UserService(client);
us.getUser("internaluser");
```

创建某种以 GitHub 支持的服务对象时，要把客户端对象传给服务的构造方法。

此外要注意，这个库专门针对 GitHub API 的第三版（v3）。如果想使用新版 API，要使用正确的 EGit 版本。如果你的企业版由于某种原因无法升级，那就不能使用这个 Java 客户端访问旧版 API。

A.6.3 JavaScript

本书使用的 JavaScript 库（GitHub.js）通过下述句法指定 GitHub 企业版后端：

```
var github = new Github({
  apiUrl: "https://github.bigdevcorp.example.com/api/v3"
  ...
});
```

A.6.4 Python

第 4 章使用的 agithub 客户端不能指定使用企业版端点创建 GitHub 客户端。如果想使用企业版端点，要定义一个继承自内置 `agithub.Github` 类的新类，然后使用这个类创建客户端。

```
class GitHubEnterprise(agithub.API):
    def __init__(self, api_url, *args, **kwargs):
        props = ConnectionProperties(
                    api_url = api_url,
                    secure_http = True,
                    extra_headers = {
                        'accept' :    'application/vnd.github.v3+json'
                        }
                    )

        self.setClient(Client(*args, **kwargs))
        self.setConnectionProperties(props)

g = GitHubEnterprise('github.mycorp.com', 'myusername', 'mypassword')
```

A.6.5 C#

Octokit 库默认连接 GitHub.com，不过也能轻易让它使用其他 API 地址。把实例化 `GitHubClient` 对象的代码换成下面这样就行。

```
var ghe = new Uri("https://github.myenterprise.com/");
var client = new GitHubClient(new ProductHeaderValue("my-cool-app"), ghe);
```

A.7 管理API

企业版有个专有的 API 端点，这是 GitHub.com 不具备的，叫作 Management Console API。这个 API 用于检查设置、维护 SSH 密钥、管理许可证，等等。可以在 Web 界面中执行的管理操作，几乎都能通过这个 API 操作（因此可以根据需求编写管理脚本）。

A.8 文档

企业版 API 的文档在这里：https://developer.github.com/v3/enterprise。

附录B

GitHub对Ruby、NodeJS（和shell）的利用

GitHub 的几位创始人深爱 Ruby 编程语言，为 Ruby 做了一些贡献，因此本书涉及 Ruby 的内容很多。

最近几年，NodeJS（服务器端 JavaScript）越来越流行，使得 JavaScript 变成一门十分吸引人的语言，因为它在客户端和服务器端都能使用。GitHub 开发了几个使用 NodeJS 编写的开源项目，十分受欢迎。

鉴于此，这篇附录再进一步说明如何使用这两门语言。

此外，熟练使用 shell 也有好处。有了 GUI 程序之后，我们很少使用命令行，但是若想充分使用 GitHub API，一定要在 shell 中使用命令行。下面这些示例都能在 bash（Bourne Again Shell）中使用，而且没有使用 bash 的高级功能（这样，如果钟爱其他 shell，就能轻易转换）。

B.1　GitHub和Ruby

讲起 GitHub 的历史，一定不能跳过 Ruby 语言。在最初操作 Git 的库中，使用 Ruby 编写的 Grit 由 Tom Preston Warner（GitHub 三个创始人之一）开发。你可以把博客免费托管在 GitHub 中，而这背后使用的工具 Jekyll 就是使用 Ruby 构建的。Gollum，即 GitHub 维基采

用的工具，是使用 Ruby 库 Grit 构建的。

若想了解 GitHub，最好懂一些 Ruby。你无需知道 Ruby 句法，安装 Ruby 后就能使用 GitHub 用到的很多工具。本书不要求你变身 Ruby 专家，但是你要能读懂 Ruby 代码。本书行文浅显易懂，只要具有基本的软件开发技能，而且精通英语，都能理解我们使用的工具。Ruby 不是完美的语言，然而却是开发者工具包中的一个有用工具，因为 Ruby 专注于提高开发者的效率。

B.1.1 安装Ruby

安装 Ruby 的方法有很多种，但是各有优缺点。作为长期用户，我知道使用预装的或包管理器中的 Ruby 有多么痛苦。一般而言，这样安装不是最好的方法。如果你不熟悉 Ruby，可以参照本附录安装，这里说明的方法最简便。

你的系统中可能已经安装了某个版本的 Ruby。Mac OS X 预装了 Ruby，多个 Linux 发行版也是如此（此外，可以使用自带的包管理器安装，快速又简单，如 apt-get）。不过，我建议你使用这里说明的方法安装，别使用系统预装的 Ruby。Ruby 包通常要求使用特定的 Ruby 版本，不过可能也支持其他版本，但是可能会遇到之前没见过的小缺陷。使用这里说明的方法可以轻易安装任何一版 Ruby，而且可以多个版本共存。使用这里说明的方法安装 Ruby 能确保使用正确的版本，而且不会干扰系统中预装的 Ruby。

我们使用 RVM 安装 Ruby。RVM 是 Ruby Version Manager 的简称，使用它可以在设备中安装多个 Ruby 版本，而且相互不冲突。为了使用书中的示例，你或许只需安装一个 Ruby 版本。使用 RVM 可以轻易安装另一个 Ruby 版本，而且不用重新配置应用，让它使用那个版本。

在不同的操作系统中使用 RVM 安装 Ruby 的方法有所不同。如果你使用 Mac OS X 或 Linux，只需在 shell 中执行下述命令：

```
$ \curl -sSL https://get.rvm.io | bash -s stable
```

这个命令会安装 RVM 和 Ruby。

如果使用的是 Windows，也可以使用 RVM 安装 Ruby，不过过程稍微复杂一些。详情参阅文档。在 Windows 中更好的安装方法是使用虚拟机，如 VirtualBox（虚拟机管理工具），把 RVM 安装到 Linux 虚拟机中。Ruby 和 RVM 对 Windows 的支持并不好，因此通常最好在 Linux 等宿主系统中安装 RVM，Ruby 对这类系统的社区支持更好。VirtualBox 和 Linux 都是免费的，无需付费就能使用（不过要花费时间）。很多带原生扩展的 Ruby gem 不能在 Windows 中编译，所以不用浪费时间考虑了，在 Windows 中应该直接使用完全免费的工具，如 VirtualBox 和虚拟机，不要苦苦寻找解决方案了。

B.1.2 重要的Ruby和RVM概念

下面是使用 Ruby 和 RVM 时的几个小提示。

- Gemfile

  Ruby 把包打包成 gem 格式。Gemfile 是清单文件，用于列出应用所需的 gem。使用 Gemfile 文件可以轻易安装所需的全部库：在 shell 终端里执行 `bundle` 命令即可。这个命令会下载所需的 gem，如果需要还会编译源码。

- .ruby-version或.rvmrc

  这两个文件告诉应用（或 shell）使用哪个 Ruby 版本。除了指定所需的依赖之外，应用中通常还会有这个文件。如果使用 RVM，它会切换到指定的 Ruby 版本，或者提示你安装。假设有个应用只能使用 Ruby 2.1.3 运行，我们可以创建一个名为 .ruby-version 的文件，写入字符串 `ruby-2.1.3`，这样启动应用时会自动使用那个 Ruby 版本。此外，其他 Ruby 工具（如无需配置的 Web 服务器 Pow）也会读取 .ruby-version 文件，使用那里指定的 Ruby 版本。

- config.ru

  这个文件的作用是使用 Rack 运行 Ruby 应用。Rack 是 Web 服务器接口，兼容多种应用服务器。如果见到 config.ru 文件，说明应用可以使用多个不同的服务器运行，例如网上很多大型网站在生成环境中使用的前端服务器，或者一台笔记本电脑中运行的小型服务器。Rack 简化了服务器的设置。

B.1.3 安装Ruby的过程中可能遇到的问题

- *缺少系统工具*

  如果使用 Mac OS X，需要安装 Xcode 和命令行工具。如果你是软件开发者，这些可能已经安装好了。如果没有，参阅在线文档安装。如果使用 Linux，可能没有安装编译器。使用这个命令可以安装全部可能用到的构建工具：`sudo apt-get install build-essential`。这个命令可能要执行一会儿，不过会把构建 RVM 和二进制 gem 所需的工具都安装上。

- *缺少开发者库*

  有些支持 Ruby 的库（例如 readline，其作用是在交互式 Ruby shell 中使用命令行历史）可能没有安装，或者对 RVM 不可用。RVM 进步很多了，现在能检测有没有所需的库，而且通常会告诉你如何正确配置这些库。使用 RVM 安装 Ruby 时，一定要阅读屏幕上的输出，找出针对所在平台的具体说明。

B.2 积极拥抱NodeJS的GitHub

NodeJS 是服务器版 JavaScript。JavaScript 是唯一在 Web 客户端普遍使用的编程语言。使用 Ruby 和 JavaScript 可以构建任何所需的 Web 应用。Hubot 这样的工具体现了使用运行在 NodeJS 平台中的 JavaScript 语言构建应用的好处。使用 NodeJS 可以构建“快速且可弹性伸缩的网络应用”。

B.2.1 安装NodeJS

nodejs.org 网站提供了不同的二进制安装程序。这通常是安装最新版 NodeJS 的最佳方式。

B.2.2 Node版本管理工具

NVM 是 Node Version Manager 的简称，类似于 RVM。与 RVM 一样，使用 NVM 可以在一台设备中安装多个 NodeJS 版本，而且可以在不同的版本之间无缝切换。这对快速迭代的 NodeJS 来说（NodeJS 的模块通常只在最新版中测试）十分有用。NVM 可以运行在 OS X 和 Linux 中。在 shell 终端里执行下述命令，安装 NVM。

```
$ curl -o- \
https://raw.githubusercontent.com/creationix/nvm/v0.25.3/install.sh | \
bash
```

这个命令会安装 NVM。安装好之后，可能要执行 `source ~/.bash_profile`，加载相关的 NVM 脚本。然后，可以执行 NVM 命令了。

```
$ nvm install 0.10 # 安装0.10版
$ nvm use 0.10     # 使用0.10版
```

NVM 提供了很多命令，详情参阅它的仓库（https://github.com/creationix/nvm）。

B.2.3 package.json

Ruby 使用 Gemfile 指定所需的库，NodeJS 也有这种文件。在 NodeJS 中，这个文件是 package.json。为了安装项目所需的全部库，要使用 `npm` 工具（使用 NVM 安装 NodeJS 时会自动安装 `npm`）。如果项目中有 package.json 文件，执行不带参数的 `npm` 命令会安装指定的全部库。如果想把库添加到现有的 package.json 文件中，可以在 `npm` 命令后面加上 `--save`；此时，`npm` 把库安装完之后会更新 package.json 文件。

B.3 命令行基础知识和shell

书中大多数章节（第 1 章除外）都只关注某一门编程语言，可是所有章节都用到了命令行。shell 有些晦涩难懂的地方你可能不熟悉，这里做些说明，再附些示例。

B.3.1 shell注释

在 shell 命令中，# 符号后面的都是注释。这样便于在同一行中说明命令的作用：

```
$ cat file.txt # 打印file.txt文件中的内容
```

这个命令在 file.txt 字符串之后结束。本附录大量使用注释说明 shell 命令的作用。

B.3.2 为命令提供变量

shell 中的进程在一个环境中运行，而这个环境可以使用键值对配置。这对键值叫作环境变量。我们经常从环境变量中读取密码，在运行时使用，而不直接在源码中编写密码。环境变量可以在命令前面使用等号连接键值对指定，也可以使用 export 命令导出，供多个命令使用：

```
$ PASSWORD=MyPwd123 myProgram  # 把PASSWORD变量的值提供给myProgram
$ export PASSWORD=MyPwd123
$ myProgram # 现在PASSWORD的值持久可用
```

B.3.3 把命令分成多行

shell 命令在按下回车键后调用。不过，有时为了可读性，想把命令分成多行写。此时，可以使用反斜线分隔各行：

```
$ git log -S http
...
$ git \
log \
-S \
http
...
```

上述命令可能并不需要分成多行，我们只是做个演示。这个示例表明，两个命令的作用完全一样。

B.3.4 把输出传给后续命令

shell 命令大都出现得很早，那时程序由多个小功能块组成，这与现在集成多项功能的 GUI 程序形成鲜明的对比。各个程序通常只做一些简单的事情，然后把信息传给另一个程序，做进一步处理。在程序之间传递数据需要一种优雅的方式，因此管道诞生了。管道增进了进程之间的通信：一个命令的输出变成另一个命令的输入。

```
$ cat /etc/mime.types | grep http
application/http
application/vnd.httphone
application/x-httpd-eruby     rhtml
```

```
application/x-httpd-php
phtml pht php
application/x-httpd-php-source      phps
```

上述命令使用 `cat` 程序输出 /etc/mime.types 文件中的内容，然后把信息传给 `grep` 程序，找出文件名中包含字符串 `http` 的全部文件。

B.3.5 重定向

类似于管道，shell 还支持使用 `>` 和 `>>` 符号把输出重定向到文件中。`>` 符号覆盖现有的文件（如果文件不存在，新建文件），而 `>>` 符号把内容添加到文件末尾。

```
$ cat /etc/mime.types | grep http > saved-output.txt
```

执行上述命令之后，saved-output.txt 文件中的文本与前面的管道示例输出的内容一样。如果这个文件已存在，那么内容会被覆盖。

作者简介

Chris Dawson 的父母是公立学校的教师。Chris 从小就对电脑十分着迷，但在学习和教学的过程中也时有挫折。他在多家著名的创业公司和科技公司（如 Apple、Virage 和 RealNetworks）默默无闻地工作过。他很感激有机会在三个大洲生活过，让他感受了多元文化的强大和活力。Chris 参与并见证了 21 世纪最激动人心的学习社区 GitHub 的发展，这让他十分享受。

Ben Straub 立志终身做一名开发者，他热衷于开发优秀的软件。他编写软件 15 年了，写过多本书，还录制过软件培训教学视频。他热爱阅读，喜欢带孩子骑车，喜欢巧克力、狗、精致的笔记本，爱好摄影、在周末修改程序、旅游、写作、做饭、手工活，醉心于好笔、Markdown，喜欢听音乐、看电影，还喜欢跟优秀的人聊天。

关于封面

本书封面上的动物是小猎犬（beagle），一种中小体型的狗（家犬）。现代的小猎犬品种于 19 世纪 30 年代在大不列颠培育，最初用于追踪小猎物，如兔子。因此，在狩猎中使用小猎犬追踪猎物就叫作“beagling”。

小猎犬属于狗这个品种里的猎犬科，但是比其他猎犬体型小，而且腿和鼻子较短。小猎犬身上通常有三种颜色（白、黑和棕），不过偶尔只有其中两种颜色。

小猎犬是备受喜爱的家养宠物，因为它们性格温顺，智商高。自伊丽莎白时代起，小猎犬就出现在大众文化中了，在莎士比亚的作品和现代的连环漫画中都有出现。

O’Reilly 出版的图书，封面上很多动物都濒临灭绝。这些动物都是地球的至宝。如果你想知道如何保护这些动物，请访问 animals.oreilly.com。

封面图片出自 Lydekker 的 *Royal Natural History, Vol. 1*。

O'REILLY®

GitHub实践

想要满足独特的开发需求？那就在下一个项目中使用GitHub提供的强大API吧！本书将手把手教你如何构建软件工具，并定制属于你的GitHub工作流程。书中每一章都要求你自己动手实践，并介绍使用GitHub提供的各项技术时应采取的折中方案以及注意事项。

如果你是经验丰富的程序员并熟知GitHub，你将学到如何使用GitHub API及相关的开源技术，如Jekyll（网站生成工具）、Hubot（NodeJS聊天机器人）和Gollum（维基）构建工具。

本书主要内容如下：

- **使用Gist API命令行工具和Ruby的API客户端Octokit，构建一个简单的Ruby服务器**
- **使用Gollum命令行工具构建一个图像管理程序**
- **使用Python构建一个搜索GitHub的GUI工具**
- **说明第三方工具和自己编写的代码如何交互**
- **使用GitHub仓库中的数据创建完整的Jekyll博客**
- **创建一个Android移动应用，读写Jekyll仓库**
- **在GitHub中托管一个完整的JavaScript单页应用**
- **使用Hubot自动审查拉取请求**

“这本书能带你入门并精通GitHub API的用法。不管你使用哪门编程语言，通过了解书中使用GitHub API开发的多个项目，你都能立即把所学的知识运用到日常工作中。”

——**Kyle Daigle**
GitHub高级平台工程师

Chris Dawson，曾就职于Apple、Virage和RealNetworks等知名IT企业，现于eBay工作。他积极参与并见证了GitHub的发展，自己也开了一家公司Webiphany。

Ben Straub，IT服务公司Gridium的开发人员，热衷于开发优秀的软件，最近刚加入GitHub。除写过多本书之外，他还兼职在线软件开发培训。

SOFTWARE DEVELOPMENT

封面设计：Randy Comer　张健

图灵社区：iTuring.cn
热线：(010)51095186转600

分类建议　计算机／软件开发

人民邮电出版社网址：www.ptpress.com.cn

O'Reilly Media, Inc.授权人民邮电出版社出版

此简体中文版仅限于中国大陆（不包含中国香港、澳门特别行政区和中国台湾地区）销售发行
This Authorized Edition for sale only in the territory of People's Republic of China (excluding Hong Kong, Macao and Taiwan)

ISBN 978-7-115-43806-5

ISBN 978-7-115-43806-5
定价：69.00元